UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

BALANCEADOR DE CARGA TOLERANTE A FALTAS BIZANTINAS

Rúben Filipe Cadima de Campos

DISSERTAÇÃO

MESTRADO EM SEGURANÇA INFORMÁTICA

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

BALANCEADOR DE CARGA TOLERANTE A FALTAS BIZANTINAS

Rúben Filipe Cadima de Campos

DISSERTAÇÃO

MESTRADO EM SEGURANÇA INFORMÁTICA

Dissertação orientada pelo Prof. Doutor Alysson Neves Bessani

2013

Agradecimentos

Começo por agradecer ao Professor Alysson Bessani pela oportunidade de trabalhar em conjunto com ele. As suas palavras positivas, ânimo e ajuda nas situações mais críticas durante este trabalho foram o mote para a sua concretização.

Em segundo lugar gostava de agradecer ao Miguel Garcia, que mais que um co-orientador, foi incansável durante o desenvolvimento do meu trabalho, não só na resolução dos meus problemas, mas principalmente na ajuda fulcral na escrita da tese, sem ele este trabalho não seria possível.

Um muito obrigado ao Hugo Sousa, que eu considero um grande amigo e companheiro, não só pela ajuda durante o desenvolvimento desta tese, mas por todos os anos do meu percurso académico.

Gostava também de agradecer a todos do laboratório 8.2.35, por proporcionarem momentos de descontração, e pelas discussões sobre todos os temas possíveis e imaginários, e um especial obrigado ao Fábio Botelho, não só em meu nome, mas também em nome da maioria das pessoas do laboratório, cujas teses talvez não tivessem sido finalizadas sem a sua ajuda.

Um muito obrigado aos meus companheiros e amigos monitores, que me acompanharam nesta aventura do ensino durante os últimos 3 anos.

Quero agradecer à minha família pelo precioso suporte durante os últimos anos, em especial, um muito, muito obrigado à minha mãe por todo o apoio que sempre me deu, e que sei que vai continuar a dar, e pelas opiniões sempre certas, que eu sempre contrariei. Obrigado.

Por último, mas não menos importante, gostava de agradecer à minha conhecida Joana Hingá pelo apoio e compreensão que nenhum outro conhecido alguma vez me dará.

Esta dissertação foi suportada pelo projeto FP7-257475 (MASSIF).

Um muito obrigado a quem não precisa de palavras.

Resumo

Middleboxes como os balanceadores de carga são elementos fundamentais nos atuais sistemas de grande escala na Internet. Tal como o nome indica, os *middleboxes* são componentes intermédios, que tipicamente fazem a ligação entre a Internet e um serviço que é prestado. A disponibilidade destes serviços está diretamente dependente da disponibilidade e da fiabilidade dos *middleboxes* que processam o tráfego. Por essa razão, torna-se necessário garantir a propriedade de tolerância a faltas nestes nós intermédios, existindo poucas contribuições neste assunto.

Neste trabalho propomos um novo modelo de *middleboxes* – em particular balanceadores de carga – que melhora significativamente as capacidades de tolerância a faltas destes componentes sem prejudicar o desempenho de forma substancial. Este modelo é baseado num novo paradigma de faltas, faltas bizantinas sistemáticas, que compreende faltas por paragem, omissão e faltas bizantinas não maliciosas.

Propomos adicionalmente um algoritmo para balanceamento de carga em *middleboxes* replicados, seguindo o paradigma do desenho inexato, que remove a sincronização entre réplicas e algumas garantias de tolerância a faltas para assegurar um desempenho equivalente às soluções sem replicação usadas atualmente.

Palavras-chave: balanceamento de carga, faltas bizantinas sistemáticas, tolerância a faltas, sistemas distribuídos, servidores web, desenho inexato.

Abstract

Middleboxes such as load balancers are fundamental elements of modern Internet-scale services. Middleboxes are components that offer a bridge between the internet and the provided service. Despite the fact that the availability of such services is directly dependent on the availability and reliability of the middleboxes handling their traffic, the techniques employed to ensure their fault tolerance are still limited.

We propose a new design for middleboxes – in particular *load balancers* – that significantly improve their fault tolerance capabilities. This design is based on a new pragmatical fault model dubbed *systematic Byzantine faults*, which encompasses crashes, omissions and even some non-malicious Byzantine faults.

Our middlebox replication algorithm follows the *inexact design paradigm*, which trades strong synchronization and output validation for a performance similar to the non-replicated solutions used in production.

Keywords: load balancing, Byzantine faults, fault tolerance, distributed systems, web servers, inexact design.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Com o crescimento das aplicações na Internet, o número de utilizadores também aumentou. Para fazer a gestão do tráfego destes clientes com as aplicações existem componentes intermédios, os *middleboxes*. Estes componentes intermédios podem ser balanceadores de carga, *firewalls* ou sistemas de detecção de intrusões e têm um papel preponderante nas infraestruturas de rede por proporcionarem capacidades como escalabilidade, disponibilidade e segurança. Estes dipositivos precisam de funcionar de forma transparente para o cliente (no máximo podem requerer mudanças ao nível da arquitetura da rede), com o mínimo de impacto nas aplicações já existentes (a velocidade de processamento deve estar próxima da velocidade da rede). Para desenvolver um *middlebox* que satisfaça os requisitos atuais é necessário que este ofereça elevada disponibilidade, escalabilidade e desempenho [26]. Disponibilidade em particular, é um assunto delicado em aplicações que oferecem serviços para utilizadores [29, 44, 49]. Por exemplo, como o Balanceador de Carga (BC) é usado para distribuir tráfego para várias aplicações e se este falhar (ou deixar de funcionar) a aplicação deixa de estar disponível. Por esta razão, o tempo de vida de um BC precisa de ser superior ao tempo de vida do serviço disponibilizado.

Os BCs mais recentes usam técnicas de replicação para garantir estes requisitos. O esquema primário-secundário é utilizado normalmente num esquema com dois BCs (um primário e um secundário), o que faz com que o sistema consiga tolerar no máximo uma falta [3, 7, 52] (figura 1.1(a)). Trabalhos mais recentes são capazes de tolerar mais faltas ao aumentar o número de réplicas e recorrendo a *middleboxes* mais complexos implementados em *software* [26, 45]. Foram também propostos *middleboxes* (mais concretamente, firewalls) baseados em protocolos de tolerância a faltas Bizantinas que oferecem elevada disponibilidade [47, 50]. A limitação principal destes protocolos está relacionada com o desempenho, fazendo com que este tipo de soluções não responda aos requisitos atuais de sistemas com um elevado número de pedidos. Esta limitação deve-se ao facto das réplicas necessitarem de uma votação sobre os pacotes (resultado de saída) e de usarem métodos criptográficos para autenticação (figura 1.1(b)). Por exemplo, um estudo recente sobre o desempenho de um *middlebox* deste tipo, com pedidos de 1Kbyte, mostra que este conse-

gue processar cerca de 27k pedidos por segundo [24], limitando os *middleboxes* que usam esta técnica a processar no máximo de 216 Mbit/s, menos de 25% da capacidade de uma rede de 1Gbit/s.

1.1 Motivação

Nesta dissertação propomos o desenvolvimento de *middleboxes* em geral, e de BCs em particular, que consigam tolerar uma grande variedade de faltas sem comprometer o desempenho. Existem duas observações principais que motivam este trabalho:

- Vários estudos mostram que faltas bizantinas não maliciosas são a principal causa de problemas graves em vários sistemas [14, 33, 41, 44]. Estes estudos indicam que as faltas tendem a ocorrer sistematicamente nos mesmos componentes e não são necessariamente críticas quando ocorrem isoladamente, mas em sistemas dentro de *datacenters*, estas faltas podem ter um efeito cascata e causar problemas em sistemas vitais (como na Amazon [1]).

- Os protocolos de rede e os protocolos de nível aplicacional oferecem várias técnicas que resolvem problemas na transmissão de dados (por exemplo, a retransmição de pacotes do protocolo *Transmission Control Protocol* (TCP)). Isto faz com que os *middleboxes* sejam candidatos interessantes para implementar métodos baseados num desenho inexato. Estes desenhos são, até certo ponto, similares ao desenho probabilístico apresentado em [10]. Neste desenho alguns erros são aceites como parte inerente do processo de computação, e as aplicações devem saber lidar com pequenos desvios da sua execução normal em troca de alto desempenho e/ou eficiência energética.

A primeira observação leva-nos a definir um novo modelo de faltas que tenha em conta as faltas descritas. Um componente que sofra uma falta bizantina sistemática apresenta regularmente comportamentos arbitrários, i.e., comportamentos que de desviam do comportamento especificado (este modelo é formalizado na secção 3.3.1). Contudo, este comportamento não se deve a ações maliciosas, mas sim a problemas acidentais que ocorrem repetidamente ao longo do tempo. Exemplos deste tipo de faltas são: a omissão da execução de uma tarefa, a corrupção do conteúdo de um pacote, ou a transmissão inesperada de mensagens.

Para detectar e remover estas faltas é necessário que o resultado de saída (dos *middleboxes*) seja verificado para que seja o mesmo em todas as réplicas, o que tipicamente requer sincronização das mesmas. Este processo cria elevados custo de desempenho, que não são aceitáveis no caso de existir a necessidade de elevadas velocidades de processamento. Por isso, seguindo o desenho inexato, a nossa solução verifica o resultado de

saída *a posteriori*, evitando a necessidade de sincronia das réplicas enquanto processam os pacotes, embora as aplicações finais possam receber resultados incorretos.

No BC, usamos uma aproximação onde as réplicas secundárias verificam as ações da réplica principal com a ajuda dos servidores (figura 1.1(c)). O papel de réplica principal e secundária (a que damos o nome de responsável e vigias) é definido para cada ligação, permitindo que a carga seja distribuída por todas as réplicas, aumentando-se assim a escalabilidade da nossa solução. Mais ainda, o BC detecta e força a remoção das réplicas com comportamentos bizantinos sistemáticos, prevenindo que estas continuem a perturbar a execução normal do sistema.

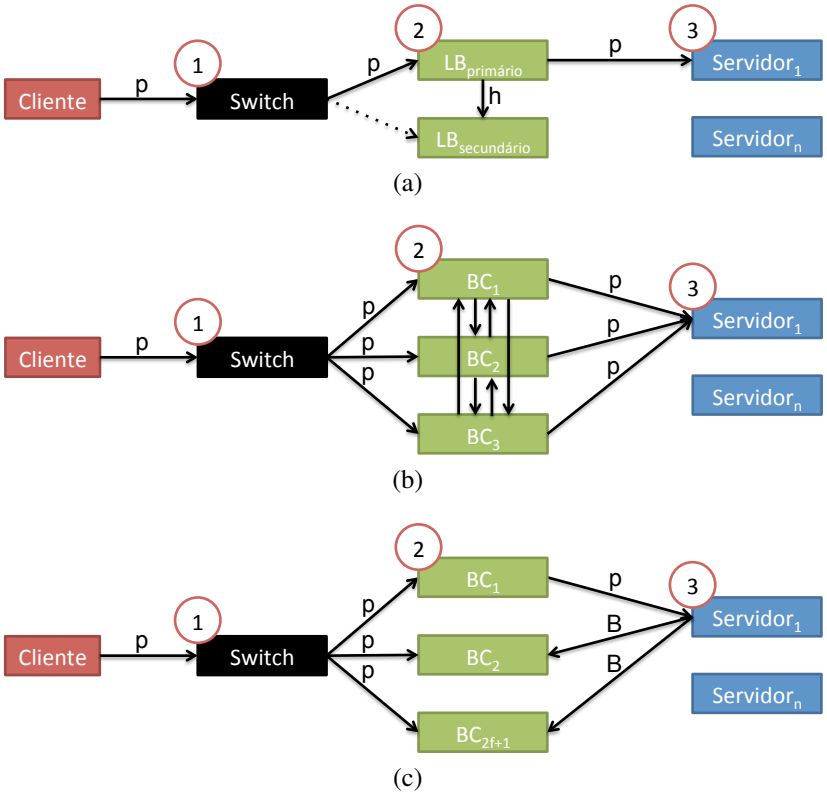

Figura 1.1: Arquiteturas tolerantes a (a) faltas por paragem; (b) faltas bizantinas; (c) faltas bizantinas sistemáticas.

1.2 Contribuições

Implementámos um protótipo de um balanceador de carga de nível 4 (nível de transporte) com algumas capacidades de firewall. O protótipo integra a monitorização das réplicas com um *switch OpenFlow* [38] para a tolerância, detecção e remoção das réplicas incorretas, com um impacto reduzido no desempenho do sistema. Realizámos experiências demonstrativas das capacidades do protótipo, sendo possível saturar uma rede de 1 Gbit/s, e esperando-se um desempenho perto dos 8 Gbit/s em redes de melhor qualidade.

Resumindo, as contribuições deste trabalho são:

1. Introdução de um novo modelo de faltas, faltas bizantinas sistemáticas não maliciosas;

2. Desenho de uma arquitetura e proposta de um algoritmo de balanceamento de carga com replicação que detecta e recupera de faltas bizantinas sistemáticas;

3. Implementação e avaliação de um protótipo de um balanceador de carga replicado, escalável e de alto desempenho, tolerante a faltas bizantinas sistemáticas. As experiências realizadas indicam que o protótipo tem capacidade para obter um desempenho similar a soluções atuais (como o *Linux Virtual Server* (LVS)), ao mesmo tempo que oferece mais garantias de tolerância a faltas.

1.3 Estrutura do Documento

O documento está organizado da seguinte forma: no capítulo 2 é feito um resumo do trabalho relacionado mais relevante; no capítulo 3 é descrito o algoritmo de balanceamento de carga tolerante a faltas e o seu comportamento em diversos cenários de faltas; no capítulo 4 é descrita a implementação da arquitetura e do algoritmo e são justificadas as diversas decisões de implementação; e no capítulo 5 é feita uma avaliação do protótipo; por último, a dissertação é concluída no capítulo 6, onde também são apresentados alguns possíveis trabalhos futuros.

Capítulo 2

Contexto e Trabalho Relacionado

Neste capítulo fazemos uma contextualização histórica sobre balanceamento de carga e descrevemos o trabalho relacionado mais relevante. Apresentamos a classificação das diferentes arquiteturas de balanceamento de carga, técnicas utilizadas e algumas políticas de distribuição de carga e a respetiva classificação. No final, apresentamos alguns balanceadores de carga utilizados atualmente e introduzimos alguns conceitos de tolerância a faltas.

2.1 Fundamentos de Balanceamento de Carga

A necessidade de fazer balanceamento de carga surgiu primeiro no contexto de energia elétrica, mais concretamente, as centrais elétricas necessitavam de alterar os fluxos de abastecimento de energia entre cidades em diferentes períodos do dia. Esta técnica foi mais tarde aplicada à computação. Esta dissertação foca-se em específico no balanceamento de carga de servidores Web.

Um sistema com servidores Web utiliza uma arquitetura igual à da figura 2.1: o cliente faz pedidos ao sistema e o servidor responde ao cliente. No caso da Web, o sistema é composto por um servidor Web que responde a pedidos *Hypertext Transfer Protocol* (HTTP), por exemplo, o *httpd* da *Apache* [4].

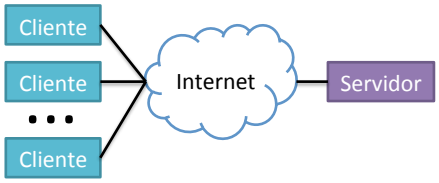

Figura 2.1: Sistema com um servidor.

Nesta arquitetura apenas um servidor recebe todos os pedidos enviados pelos clientes. Se o número de clientes aumentar, o servidor pode atingir o seu limite de processamento e deixa de conseguir responder a alguns pedidos, fazendo com que o sistema pareça indisponível para os clientes. O mesmo acontece se o servidor simplesmente parar (por

exemplo, por falta de energia). Com cenário surge a necessidade de utilizar um BC para distribuir os pedidos enviados pelos clientes para diferentes servidores Web (figura 2.2).

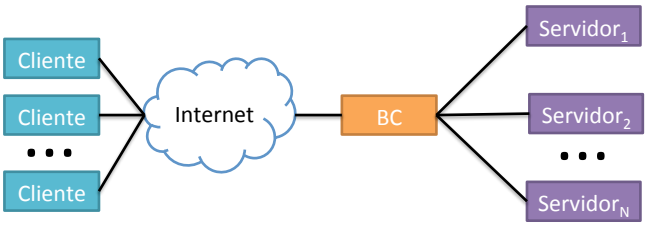

Figura 2.2: Sistema com um BC e vários servidores.

Numa arquitetura de balanceamento de carga podemos distinguir três componentes principais:

1. o cliente, que faz os pedidos ao sistema;
2. o servidor, que responde aos pedidos dos clientes; e
3. o BC, que encaminha os pedidos para o servidor escolhido.

Os BCs em específico podem ser decompostos em arquitetura e política. A arquitetura especifica o desenho da rede e a técnica de balanceamento de carga usada (secção 2.1.1) e a política especifica como é escolhido o servidor que responde ao pedido do cliente (secção 2.1.2). Uma técnica de balanceamento de carga define como é que um pedido passa do BC para o servidor (por exemplo, o pedido é retransmitido sem alterações para o servidor ou o endereço *Internet Protocol* (IP) de destino é alterado antes do pacote ser encaminhado). Um pedido é, em regra geral, equivalente a um pacote, com exceção para pedidos cujo tamanho ultrapasse os 1500 bytes (tamanho máximo típico de um pacote IP), neste caso o pedido é fragmentado em vários pacotes. Este pedido é tipicamente uma requisição de uma página Web (ficheiro no servidor) ou um pedido a uma aplicação (por exemplo, atualização de uma base de dados).

2.1.1 Arquiteturas de Balanceadores de Carga

Existem várias classificações para os BCs, dependendo de:

- **Distribuição do sistema.** Se os componentes do sistema estão na mesma rede privada, consideramos que o sistema é distribuído a nível local. Se por outro lado os componentes do sistema estão distribuídos geograficamente, o sistema é distribuído a nível global.
- **Tipo de arquitetura do sistema.** Em arquiteturas de *uma via* os servidores respondem diretamente aos clientes (apenas o pedido passa pelo BC). Em arquiteturas de *duas vias* as respostas dos servidores (tal como os pedidos dos clientes) passam primeiro pelo BC.

- **Camada de *Open Systems Interconnection* (OSI) [27].** Tal como a pilha protocolar OSI, a classificação do BC varia entre um e sete. As arquiteturas mais usadas são: *(1)* as de nível transporte (nível 4) – a política de distribuição de carga apenas têm em conta os cabeçalhos TCP/IP; e *(2)* as de nível aplicação (nível 7) – a política de distribuição de carga consegue também aceder ao conteúdo dos pacotes.

De seguida apresentamos as técnicas que podem ser usadas nas diferentes arquiteturas de balanceamento de carga. Este tema é aprofundando em [21, 30].

Arquiteturas Distribuídas a Nível Global

Em sistemas distribuídos globais o balanceamento de carga pode ser feito apenas por: *(1)* servidores de *Domain Name Server* (DNS) [20] ou *(2)* servidores Web (ver figura 2.3).

Com servidores de DNS, o balanceamento de carga ocorre quando um cliente pede a tradução de um nome (*Uniform Resource Locator* (URL)) para um endereço IP. O servidor de DNS tem vários endereços IP (dos vários servidores) que correspondem a esse nome. Isto faz com que o resultado da tradução do nome sejam IPs diferentes para diferentes clientes. O sistema de DNS funciona como uma cadeia com vários níveis. Um cliente pede uma tradução a um servidor de DNS, por exemplo do nível 3, e se este servidor não conseguir traduzir o nome, pede a tradução a um servidor de DNS de nível 2 (e assim sucessivamente até ao servidor de DNS autoritativo – de nível 0). O servidor de DNS autoritativo é o único que sabe sempre todos os IPs correspondentes a um dado nome. O problema com esta técnica é que os servidores de níveis mais altos guardam as respostas temporariamente após o primeiro pedido de tradução, então os pedidos seguintes vão ser traduzidos por estes servidores para o mesmo endereço IP e deixa de existir balanceamento de carga.

Com servidores Web, existem três técnicas que permitem fazer balanceamento de carga:

1. *Tringulation*. Nesta técnica, se um servidor não conseguir responder ao pedido, este encapsula o pacote IP do cliente noutro pacote IP e encaminha-o para outro servidor [13].

2. *HTTP redirection*. Esta técnica faz uso dos códigos 301 (movido) e 302 (encontrado) do HTTP para obrigar o cliente a fazer o pedido a outro servidor [15].

3. *URL rewriting*. Esta técnica substitui os URL da página pedida pelo cliente por URL gerados dinamicamente que são traduzidos pelos servidores de DNS em outros endereços IP.

Contrariamente ao balanceamento de carga usando o DNS, estas técnicas têm a desvantagem de consumir os recursos de um servidor, o que aumenta os tempos de resposta aos pedidos.

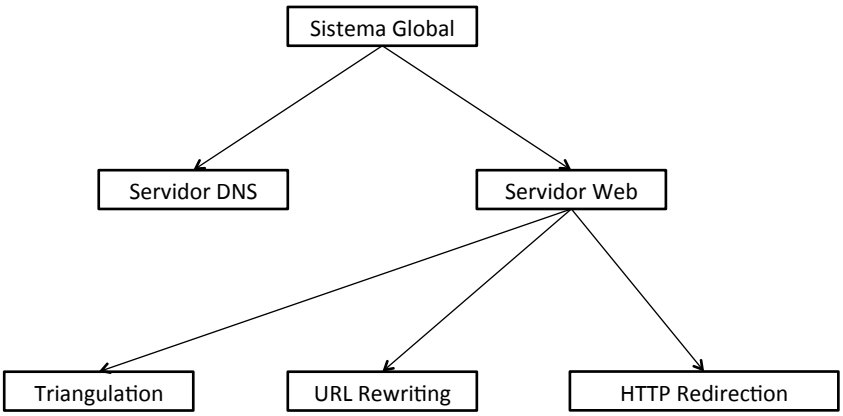

Figura 2.3: Técnicas de balanceamento de carga em sistemas globais.

Balanceamento de Carga Local de Nível 4

O balanceamento de nível 4 é feito ao nível dos protocolos de rede (protocolo IP). Depois da ligação ser estabelecida e do BC receber um pacote, este verifica numa tabela de encaminhamento qual o seguimento a dar ao pacote. Esta tabela faz corresponder uma ligação a um servidor. As técnicas de balanceamento de carga dividem-se em (ver figura 2.4):

Uma via. Os pacotes IP do cliente passam pelo BC antes de chegarem ao servidor, mas os pacotes de resposta são enviados diretamente para o cliente. Existem três técnicas diferentes:

- *Packet single-rewriting* [25]. Esta técnica substitui o endereço IP de destino dos pacotes do cliente pelo endereço IP do servidor, recalcula os *checksum* e encaminha o pacote para o servidor.

- *Packet tunneling*. Esta técnica encapsula o pacote IP noutro pacote IP com o endereço de destino do servidor. Depois, encaminha o pacote para o servidor de destino. O servidor desencapsula o pacote e responde ao cliente. A desvantagem desta técnica é que obriga os servidores a suportar *IP tunneling*.

- *Packet forwarding* [19]. Esta técnica substitui os endereços *Media Access Control* (MAC) de destino dos pacotes pelo endereço MAC do servidor e encaminha o pacote para o servidor. Para utilizar esta técnica é necessário desativar o protocolo de tradução de endereços *Network Address Translation* (NAT) porque os servidores têm todos o mesmo IP, o que pode causar colisões.

Duas vias. Os pacotes IP do cliente e do servidor passam sempre pelo BC. Nestas arquiteturas apenas existe a técnica *packet double re-writing*. Esta técnica altera os cabeçalhos IP, recalcula os *checksums* e encaminha os pacotes (do cliente e do servidor) para o respetivo destino.

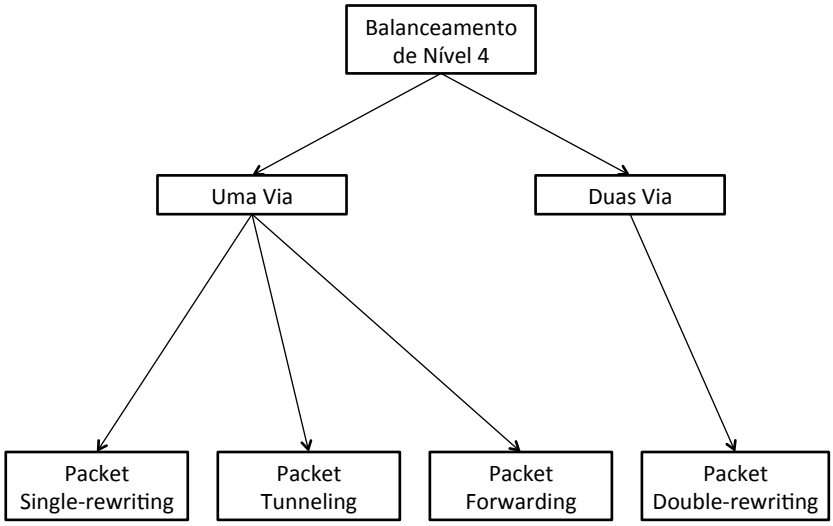

Figura 2.4: Técnicas de balanceamento de carga de nível 4 em sistemas locais.

Balanceamento de Carga Local de Nível 7

Para este tipo de arquiteturas descrevemos apenas as que consideramos mais relevantes, no entanto todas as técnicas que conhecemos [21, 30] estão apresentadas na figura 2.5. O balanceamento de carga de nível 7 é feito ao nível aplicacional, e permite o uso de políticas de distribuição mais complexas porque dá acesso ao conteúdo dos pacotes. As técnicas de balanceamento de carga dividem-se em:

Uma via. Existem duas técnicas principais usadas em arquiteturas de uma via:

1. *TCP Gateway de uma via* [37]. Esta técnica obriga o BC a ter uma ligação aberta com todos os servidores por onde são encaminhados todos os pedidos dos clientes.

2. *TCP Rebuilding* [36]. Esta técnica estabelece uma ligação entre o cliente e o BC e quando o cliente envia o pedido para o BC, este encaminha-o para o servidor que reconstrói a ligação (alterando os números de sequência e *acknowledge*) e responde ao pedido do cliente.

Duas vias. A técnica principal nas arquiteturas de duas vias é o *TCP gateway*. Esta técnica obriga o BC a manter uma ligação persistente[1] com todos os servidores. Quando

[1]Uma ligação persistente permite que sejam enviados vários pedidos HTTP pela mesma ligação.

o BC recebe um pacote do cliente, encaminha-o para o servidor através dessa mesma ligação. A resposta do servidor, que é enviada pela mesma ligação, chega ao BC e este reencaminha-a para o cliente. Esta é a técnica usada pelo *httpd-bc* da *Apache* [11].

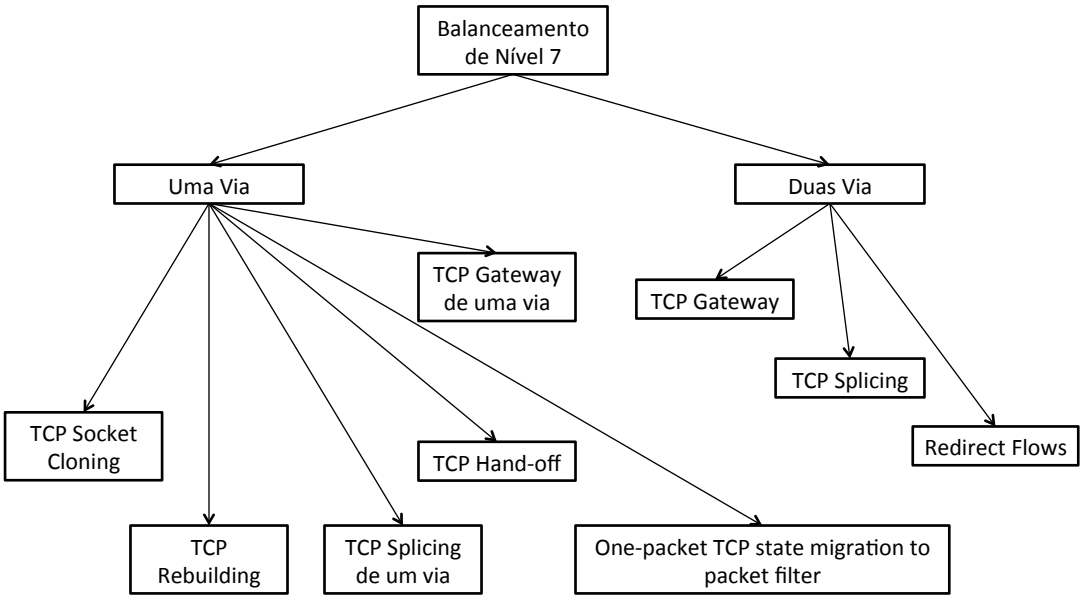

Figura 2.5: Técnicas de balanceamento de carga de nível 7 em sistemas locais.

2.1.2 Políticas de Distribuição de Carga

As políticas de distribuição de carga são usadas pelo BC para decidir qual o servidor a que se destina o pedido do cliente. Existem políticas estáticas ou dinâmicas e políticas com ou sem conhecimento de conteúdo dos pacotes.

Políticas Estáticas vs. Políticas Dinâmicas

As políticas estáticas tomam sempre as mesmas decisões, i.e., não têm em conta o estado das aplicações ou servidores. Uma dessas políticas é o *Round-Robin*. Esta política distribuí uniformemente os pedidos dos clientes pelos servidores. Por exemplo, num sistema com dois servidores, o primeiro pedido seria entregue ao primeiro servidor, o segundo pedido ao segundo servidor, o terceiro pedido ao primeiro servidor, o quarto pedido ao segundo servidor, e assim sucessivamente. Existe uma variante desta política que atribui pesos aos servidores (*Weighted Round-Robin*) e em vez de distribuir 50% dos pedidos para cada um, distribui uma percentagem adequada ao seu peso. Por exemplo, se o servidor 1 tem um peso de 0.7 (por exemplo, porque tem um processador melhor) e o servidor 2 um peso de 0.3, 70% dos pedidos serão entregues ao servidor 1 e 30% ao servidor 2.

As políticas dinâmicas têm em conta o estado dos servidores e dos clientes (muitas destas políticas analisam também o conteúdo do pedido – ver secção 2.1.2). Por norma,

os BCs que utilizam estas políticas comunicam constantemente com os servidores para saber o seu estado, ou guardam uma aproximação do estado dos servidores através dos pedidos que foram encaminhados. O estado dos servidores pode ser, por exemplo, o número de ligações que cada servidor tem abertas, assim o BC pode encaminhar o próximo pedido para o servidor com menos ligações. Exemplos de políticas dinâmicas são: *Least Loaded*, *Least Connections* (usada pelos balanceadores de carga da *Cisco* [6] e da *F5 Networks* [7]), *Fastest Response Time* (usada nos balanceadores de carga da *Foundry Networks* [8]) e *Round Robin* dinâmico [32].

As políticas estáticas são mais rápidas a executar mas podem fazer más escolhas e encaminhar pedidos para servidores sobrecarregados. Por sua vez, as políticas dinâmicas demoram mais tempo a executar mas fazem melhores escolhas (porque têm conhecimento do estado dos servidores). Assim, por vezes, os sistemas que usam políticas dinâmicas têm melhor desempenho que os sistemas que usam políticas estáticas.

Conhecimento do Conteúdo dos Pacotes

As políticas que têm acesso ao conteúdo dos pedidos usam essa informação para fazer uma escolha melhor sobre qual deve ser o servidor responsável pelo pedido. São, em regra geral, políticas dinâmicas com conhecimento do estado dos servidores e só podem ser usadas em arquiteturas de balanceamento de carga de nível 7 (nível aplicação). Tipicamente, estas políticas tentam melhorar o desempenho através da redução dos acessos ao disco nos servidores (encaminham pedidos da mesma página Web para os mesmos servidores que a devem ter em *cache* na memória) ou uso de servidores especializados (por exemplo, o servidor 1 responde a pedidos HTTP e o servidor 2 responde a pedidos de *streaming* de vídeo).

A política *Size Interval Task Assignment with Equal Load* (SITE-E) [31] é um exemplo de uma destas políticas, na qual o BC analisa o conteúdo do pacote e verifica qual a página Web que o cliente está a pedir. Esta política distribui os pedidos de acordo com o tamanho das páginas Web, por forma a uniformizar o trabalho de cada servidor. Outro exemplo é a *Client Aware Policy* (CAP) [22], que encaminha os pedidos dos mesmos clientes sempre para os mesmos servidores (*client-server afinity*).

A *Locality-Aware Request Distribution* (LARD) [42], por sua vez, é uma política mais complexa. Esta política analisa a utilização de recursos dos servidores, e enquanto um servidor não estiver no limite de utilização, será sempre o mesmo a responder a todos os pedidos. Adicionalmente, esta política mantém uma tabela que mapeia as páginas Web mais acedidas para um ou mais servidores, reduzindo desta forma os acessos a disco ao obrigar sempre os mesmos servidores a responder aos pedidos de páginas que já têm em *cache*. A política *Workload-Aware Request Distribution* (WARD) [18] verifica que páginas Web são mais acedidas e obriga todos os servidores a ter uma réplica dessa página. Desta forma todos os servidores podem responder aos pedidos das páginas mais requisitadas.

2.2 Alguns Balanceadores de Carga Usados Atualmente

2.2.1 LVS

O LVS [52] é uma implementação *opensource* de um BC num sistema Linux, desenvolvido para garantir escalabilidade e disponibilidade. O LVS oferece várias políticas estáticas e dinâmicas[2] para serem usadas em conjunto com três técnicas de balanceamento de carga:

- **NAT.** Equivalente ao *packet double re-writing*, reescreve o pedido antes de o encaminhar para o servidor de destino e reescreve a resposta antes de a encaminhar para o cliente. Não é necessário estabelecimento de uma ligação TCP entre o BC e os servidores, mas os servidores devem estar configurados para responder diretamente ao BC e não ao cliente.

- ***IP tunneling*.** O BC encapsula os pedidos do cliente dentro de outro pacote e envia para o servidor, que ao receber o pacote desencapsula-o, analisa o pedido original do cliente, e responde diretamente ao cliente. Esta técnica requer que os servidores suportem *IP tunneling*.

- ***Direct routing*.** Equivalente ao *packet forwarding*, é a técnica mais eficiente porque o BC apenas tem que substituir o endereço MAC de destino do pacote recebido do cliente pelo do servidor de destino. O servidor pode responder diretamente ao cliente.

2.2.2 *WebSphere*

Este balanceador de carga desenvolvido pela IBM [34] implementa uma arquitetura semelhante ao LVS e também usa as técnicas de *direct routing* e de NAT. Este sistema tem como objetivo principal oferecer alta disponibilidade e fornecer funcionalidades como *client-server afinity*; personalização de regras de distribuição de carga; e analisar o conteúdo dos pacotes; etc.

2.2.3 *Apache httpd-bc*

O *httpd-bc* é um servidor Web *opensource* que pode ser usado como BC através de módulos externos, mais concretamente, o módulo *Proxy* [11] e o módulo *Connectors* [12]. A técnica de balanceamento usada por este BC é a *TCP gateway* (o BC tem uma ligação persistente com todos os servidores e encaminha os pedidos do servidor através dessas ligações). Esta implementação oferece algumas políticas de balanceamento de carga dinâmicas e facilita a integração de novas políticas.

[2] http://kb.linuxvirtualserver.org/wiki/Category:Job_Scheduling_Algorithms

2.3 Tolerância a Faltas em Balanceadores de Carga

Embora existam várias implementações em uso, e várias propostas na literatura, não existem muitos trabalhos que estudem e desenvolvem BCs no contexto de tolerância a faltas bizantinas. Existem alguns trabalhos com propostas de BCs tolerantes a faltas por paragem [11, 52], no entanto, não existem propostas para o cenário de faltas bizantinas.

Tipicamente, este tipo de faltas são tratadas e mascaradas através de replicação, mas necessitam que seja executado um protocolo (além do protocolo que cada réplica executa numa situação normal) de replicação que especifica o comportamento de cada réplica no caso de falta.

Os componentes incorretos, quando sofrem uma falta por paragem, deixam de executar os protocolos especificados. As faltas bizantinas [35], por sua vez, fazem com que o componente incorreto se comporte de forma arbitrária, i.e., o componente incorreto pode, por exemplo, corromper ou criar mensagens, corromper o estado do sistema ou simplesmente deixar de executar (equivalente a uma falta por paragem).

As soluções para as faltas por paragem obrigam a uma replicação $f + 1$ porque para tolerar este tipo de faltas é suficiente ter uma réplica correta para tolerar f faltas (por exemplo, para sistema tolerar três faltas por paragem em simultâneo são precisas quatro réplicas). Existe uma comunicação constante entre réplicas, quando uma réplica detecta (porque a comunicação parou) que a réplica mestre falhou, substitui a réplica e passa a ser a nova mestre. Todos os BCs apresentados na secção 2.2 toleram faltas por paragem usando esta abordagem.

Existem trabalhos recentes que propõem novas formas de garantir elevada disponibilidade em *middleboxes* como os balanceadores de carga [26, 45]. O *Ananta* é um balanceador de carga em *software* e é usado na nuvem *Azure* da *Microsoft* [26]. O objetivo principal do *Ananta* é garantir que o sistema escale infinitamente, por isso apenas implementa mecanismos simples de detecção a faltas (o típico $1 + f$ réplicas, ver secção 2.3). O *Pico* é um trabalho recente que oferece elevada disponibilidade [45]. O objetivo principal do *Pico* é que a sua replicação seja eficiente de modo a que a transição de ligações para outra réplica (no caso de uma réplica falhar) seja simples e com o mínimo de perda de pacotes. O *Pico* usa um controlador *OpenFlow* [38] (tal como a nossa solução, ver capítulo 4) para gerir a replicação e a detecção das réplicas incorretas.

Por sua vez, as propostas que toleram faltas bizantinas utilizam também replicação, mas neste caso necessitam de $3f + 1$ réplicas. Devido a certos resultados de impossibilidade [35], não é suficiente ter apenas uma réplica correta ($f + 1$) e nem uma maioria de réplicas corretas ($2f+1$) usadas pela replicação tolerante a faltas bizantinas convencional. Estas soluções [16, 23, 47, 50] assumem forte sincronia entre réplicas (o que leva a uma elevada comunicação) para que a decisão final de todas as réplicas seja a mesma. Isto faz com que o desempenho destas soluções seja muito abaixo da capacidade das redes atuais.

2.4 Sumário

Este capítulo apresentou alguns conceitos fundamentais e trabalhos relacionados que consideramos relevantes para esta dissertação. Descrevemos as arquiteturas de sistemas de balanceamento de carga. Apresentámos algumas políticas de distribuição estáticas e dinâmicas, e alguns balanceadores de carga utilizados em diferentes cenários. Por fim, descrevemos as abordagens para tolerância a faltas em *middleboxs*. De todas as arquiteturas, o nosso trabalho utiliza uma arquitetura de nível 4 (nível TCP/IP) e a técnica *packet-forwarding* para encaminhar os pacotes para os servidores. O nosso trabalho possibilita ainda o uso de políticas estáticas e dinâmicas sem conhecimento do conteúdo dos pacotes. No próximo capítulo descrevemos um algoritmo tolerante a faltas bizantinas sistemáticas para balanceadores de carga.

Capítulo 3

Balanceador de Carga Tolerante a Faltas Bizantinas

Neste capítulo começamos por descrever o desenho inexato do nosso algoritmo, seguido pelo modelo e propriedades do sistema. Depois, descrevemos detalhadamente o algoritmo de tolerância a faltas bizantinas sistemáticas e finalizamos com uma descrição dos potenciais problemas e soluções do algoritmo e dos vários cenários de faltas bizantinas toleradas pelo algoritmo.

3.1 A Necessidade de um Novo Modelo de Faltas

Na última década, vários estudos mostraram que os componentes de *hardware* e *software* apresentam mais comportamentos bizantinos do que inicialmente esperado, e que mesmo as técnicas utilizadas, como por exemplo o uso de *checksums* para detectar a corrupção de mensagens, não são suficientes para detectar todos os tipos de faltas (ver [24] para uma descrição mais detalhada). Alguns destes estudos confirmam que os erros aparecem com alta probabilidade em:

- *Memórias:* Um estudo a servidores de *datacenters* mostra que uma grande maioria (65-82%) dos bancos de memórias sofrem de múltiplos erros (nos mesmos endereços ou em endereços vizinhos) [33]. Este estudo mostra também que a distribuição dos erros não é uniforme: apenas 10% dos endereços de memória são responsáveis por 90% dos erros observados. Estes estudos provam que as faltas sistemáticas são mais prováveis de acontecer do que as faltas esporádicas. Estas observações são confirmadas noutro estudo feito em computadores pessoais [41].

- *Discos:* Outro estudo sobre corrupção de dados em armazenamento em discos observou num período de 41 meses mais de 400k *checksums* errados [14]. De forma semelhante ao que foi observado para as memórias, estas faltas não são independentes: a probabilidade de um sistema de armazenamento experienciar um segundo

erro no *checksum* após a primeira ocorrência ronda os 60%, enquanto que a probabilidade de um disco ter um primeiro erro nos *checksums* é de apenas 0.7%. Tal como nas memórias, a distribuição dos erros em discos tem uma cauda muito grande.

- *Computadores Pessoais:* Outro estudo realizado por investigadores da Microsoft evidencia que as faltas em computadores pessoais não são esporádicas nem independentes [41]. Um máquina que falhe por paragem uma vez, tem uma probabilidade de 30% de voltar a falhar, enquanto que a probabilidade de uma máquina falhar pela primeira vez é de 0.5%.

Resumindo, estes estudos mostram que um pequeno número de componentes é responsável por um grande número de faltas e que a probabilidade de ocorrerem faltas sistemáticas é elevada. Por isso, o nosso objetivo é detectar e remover estes componentes incorretos do sistema.

3.2 Desenho Inexato

Um desafio atual conhecido é sustentar a lei de Moore mesmo com as limitações do silício e os requerimentos ao nível do consumo de energia. Para responder a este desafio têm sido adoptados sistemas baseados num desenho inexato (ou probabilístico) que troca alguma qualidade de resultados por um melhor desempenho e consumo de energia [10, 43]. Estes sistemas exploram as características do meio que pode ocasionalmente gerar alguns erros nas aplicações. Naturalmente, existem aplicações em que não é possível usar este tipo de sistemas, mas outras, como as relacionadas ao processamento de sinais digitais, podem ser facilmente adaptadas para tolerar erros esporádicos [28].

Por exemplo, se considerarmos comunicações HTTP/HTTPS, existem vários mecanismos utilizados que lidam com os problemas da rede: *(1)* o IP e o TCP incluem *checksums* nos pacotes que são verificados pelo receptor. Um pacote é descartado se os *checksums* não estão corretos; *(2)* o TCP retransmite os pacotes perdidos passado algum tempo e, adicionalmente, se um pacote TCP for enviado para um destino em que não exista uma ligação válida, o pacote é descartado; *(3)* o *Secure Sockets Layer* (SSL) assegura todos os meios necessários para um controlo de segurança sobre o TCP, incluindo integridade através do uso de autenticadores de mensagens (do inglês: *message authentication code* – MAC)[1]; *(4)* o HTTP/HTTPS inclui um conjunto de códigos de erro que permitem às aplicações solicitar retransmissões.

Todas estas funcionalidades podem ser exploradas para implementar um mecanismo de tolerância a faltas bizantinas sistemáticas sobre um desenho inexato de um *middlebox*, como veremos em seguida.

[1] Os MACs são também chamados de funções de *hash* criptográficas. Esta função recebe como entrada uma mensagem e uma chave e dá como resultado um código (MAC). Com este código é possível apenas a quem tiver a chave verificar a autenticidade e integridade da mensagem.

3.3 Modelo de Sistema

O nosso sistema é composto por um número ilimitado de clientes, um conjunto $\mathcal{R}$ de réplicas do BC, e um conjunto $\mathcal{S}$ de servidores responsáveis por processar os pacotes. Assumimos que existe um *switch* e um controlador confiáveis para disseminar o tráfego para o subconjunto das réplicas corretas de $\mathcal{R}$. Todos os clientes apenas conhecem um endereço IP (endereço do serviço) para o qual enviam os pedidos HTTP. Os pacotes do pedido são enviados através de uma ligação TCP que, como o protótipo do BC é de nível 4 de uma via, é estabelecida entre o cliente e o servidor.

3.3.1 Modelo de Faltas

O balanceador de carga pode falhar de forma bizantina [35], i.e., uma réplica pode, de alguma forma, desviar-se do seu comportamento especificado adoptando um comportamento aleatório. Contudo, não consideramos que um componente incorreto possa ser controlado por um atacante. Adicionalmente, assumimos $|\mathcal{R}| \geq 2f + 1 + k$, sendo f o limite máximo de réplicas incorretas em simultâneo e k o número de réplicas secundárias usadas para melhorar a escalabilidade do sistema.

Assumimos que réplicas incorretas tendem a exibir um comportamento bizantino de forma sistemática, i.e., se uma réplica é incorreta, ela comporta-se de forma incorreta sistematicamente. É de salientar que as faltas por paragem estão incluídas nestas faltas porque quando uma réplica falha não volta a executar. Assumimos também que o comportamento arbitrário é não malicioso. Esta assunção, que está de acordo com os estudos mencionados anteriormente que mostram que existe uma correlação temporal e espacial na localização das faltas, permite-nos de forma eficiente detectar componentes incorretos sem usar mecanismos dispendiosos da tolerância a faltas bizantinas normal.

3.3.2 Modelo de Sincronia

Assumimos que todos os componentes têm acesso a relógios aproximadamente sincronizados (acedidos através de $\mathcal{T}_{now}$). Assumimos também que o processamento dos pacotes e a comunicação entre dois processos está limitada por Δ_p e Δ_c, respetivamente. Apesar destes limites terem uma grande probabilidade de ser verificados em *datacenters* [9], o nosso desenho inexato consegue tolerar violações destes limites (ver secção 3.6.2).

Assim, é possível implementar um algoritmo de balanceamento de carga tolerante a faltas bizantinas sistemáticas, com um bom desempenho, em que as réplicas não precisam necessariamente de ter a mesma configuração em todos os instantes (desenho inexato), mas algures no tempo todas convergem para a mesma configuração.

3.3.3 Propriedades

O nosso algoritmo de replicação requer que a política de distribuição de carga seja determinística, i.e., se uma réplica correta seleciona um servidor s como destino para um dado pacote, todas as outras réplicas corretas também selecionam o mesmo servidor como destino para esse pacote. O acesso a esta política é feito através de duas funções genéricas: *getDestination()* para obter um servidor de destino para o pacote; e *updatePolicy()* para atualizar a política (por exemplo, informar a adição/remoção de novos servidores). Assumimos que as políticas de distribuição não consideram estado, i.e., não têm em conta decisões anteriores para decidir qual o destino do pacote; ou apenas mantêm estado para cada ligação, i.e., o destino de um pacote depende apenas de pacotes anteriores na mesma ligação. Se for necessário um estado global do sistema, podem ser usadas outras técnicas [45, 46] para implementar uma política que tenha em conta a sincronização de estado entre réplicas, mas neste trabalho não consideramos tais políticas. Para além disto, o algoritmo de replicação é completamente independente da política de distribuição usada pelo BC.

O nosso algoritmo para um BC confiável satisfaz duas propriedades principais, relacionadas com a tolerância e detecção de faltas:

1. Se um pacote de um cliente for recebido por $f + 1$ réplicas corretas, algures no tempo será entregue ao servidor de destino;

2. Se uma réplica se comportar incorretamente de forma sistemática, esse comportamento vai ser detectado algures no tempo e a réplica será removida.

É de salientar que estas propriedades estão descritas em condições de *"se... então..."*, que estão de acordo com o nosso desenho inexato, e estas propriedades apenas são garantidas se o sistema se comportar da forma esperada com grande probabilidade.

3.4 Descrição Geral da Arquitetura

A figura 3.1 apresenta os componentes principais do balanceador de carga proposto, e mostra que o BC é composto por um *switch*, um controlador e um conjunto de réplicas do BC. Para além disso, a figura mostra que existe um conjunto de clientes que enviam pacotes para um conjunto de servidores atrás do BC. O *switch* dissemina todo o tráfego, destinado ao endereço IP do serviço, para todas as réplicas do BC (1). Depois, para cada pacote recebido, uma das réplicas é escolhida como responsável (2_a), que irá encaminhar o pacote para um servidor (3). Existe um subconjunto de réplicas do BC que são escolhidas como vigias (2_b), que mais tarde verificam se o pacote foi corretamente encaminhado pelo responsável. Esta verificação é feita com base nos *bags* (esta estrutura de dados será explicada na secção 3.5.1) recebidos dos servidores com os pacotes processados na ronda

anterior (4). Uma ronda corresponde ao tempo que um servidor utiliza o *bag* antes de o enviar para as réplicas. Se os vigias detectarem que um pacote não foi corretamente encaminhado para um servidor, esse pacote é retransmitido. O controlador é informado pelos vigias se uma réplica se comportar de forma incorreta sistematicamente, e essa réplica é depois removida do sistema (5).

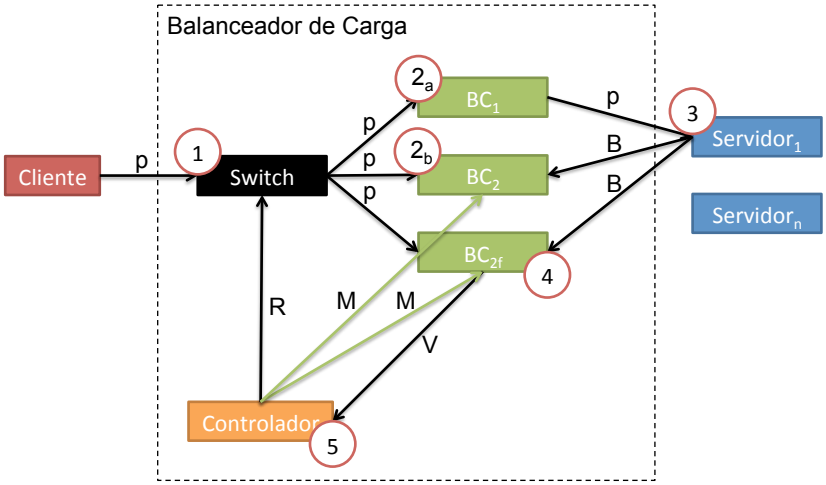

Figura 3.1: Descrição geral da arquitetura do nosso balanceador de carga.

3.5 Algoritmos

Nesta secção descrevemos o algoritmo de replicação tolerante a faltas bizantinas sistemáticas para BCs. O algoritmo está dividido em três módulos, cada um com diferentes procedimentos.

3.5.1 Balanceador de Carga

O algoritmo 1 descreve o comportamento de uma réplica i do balanceador de carga. Quando a réplica inicia (ou recupera de uma falta), inicializa cinco tabelas com n entradas, uma para cada réplica do BC:

1. $Pend$ armazena todos os pacotes encaminhados que ainda não foram verificados;
2. $susp$ e $asusp$ mantém o número de suspeições totais e suspeições ativas para cada réplica, respetivamente;
3. $faulty$ indica se uma dada réplica é considerada incorreta;
4. $ignore$ contém um contador (decrescente) do número de rondas em que nenhuma réplica deve ser considerara incorreta (para simplificar, esta variável é inicializada com um valor predefinido `IGNORE`).

Depois, a réplica envia uma mensagem `HELLO` para o controlador a avisar que está pronta para entrar no sistema (linhas 1-6).

Quando um pacote r chega à réplica (linha 7), são executados os seguintes passos. Primeiro a réplica determina qual é a réplica responsável por encaminhar o pacote, aplicando uma função de *hash* sobre o endereço e porto de origem (representado por $\mathsf{H}(r.src)$) para obter o identificador, e desta forma distribuir as ligações por todas as réplicas do BC. É de salientar que esta estratégia de distribuição de ligações por todas as réplicas simplifica a gestão de ligações em políticas de distribuição que necessitem de estado. Se a réplica responsável é incorreta, as réplicas seguintes são escolhidas, sequencialmente, para serem responsáveis até uma réplica correta ser encontrada (linhas 9-10). A réplica também escolhe um conjunto de $2f$ réplicas para servirem de vigias para aquele pacote (linha 11).

Se a réplica é a responsável ou é um dos vigias, usa uma política de distribuição determinista, através da função *getDestination()*, para selecionar o servidor $dest$ que irá processar o pacote (linhas 12-13). Podem ser adicionadas capacidades de *firewall* ao BC, onde regras simples podem especificar os pacotes que podem ou não ser encaminhados (por exemplo, os pacotes não são encaminhados se o endereço IP de destino não corresponder ao endereço do serviço). Neste caso a função *getDestination()* devolve $\perp$ e o pacote é descartado (linha 14). O responsável encaminha o pacote para o servidor final. São guardados numa entrada na tabela $Pend$ correspondente à réplica responsável uma cópia do pacote, o instante em que foi recebido e o servidor selecionado (linhas 15-18).

Os vigias validam as ações do responsável $forw$ quando recebem um *bag* do servidor $dest$ (linha 19). O *bag* contém informação sobre quais os pacotes que foram encaminhados na ronda anterior pelo responsável $forw$ para o servidor $dest$. Para cada pacote na tabela de pendentes (conjunto $Pend[forw]$), a réplica verifica se estes foram corretamente recebidos pelo servidor. Em caso afirmativo, o pacote é removido do conjunto (e do *bag*) e é verificado o próximo pacote (linhas 23-25). Caso contrário, se o pacote foi recebido pelo BC há já algum tempo, o responsável é considerado suspeito (atualizando a variável $suspect$ para verdadeiro). Adicionalmente, uma cópia do pacote é enviada para o servidor para garantir que este é processado (linhas 26-29). O limite `TIMEOUT` decide quando é que um vigia considera um responsável suspeito por não ter encaminhado um pacote. Este valor deve ter em conta os tempos de transmissão de pacotes na rede local (*round trip time*) e o tempo de processamento de um servidor (visto que os *bags* demoram algum tempo até serem enviados para os vigias). Consequentemente o `TIMEOUT` deve ser maior que $2\Delta_c + \Delta_p$, mas normalmente deve ser muito maior porque os servidores enviam *bags* compactados para verificação (ver secções 3.5.2 e 3.6.1). Por isso, é necessária alguma afinação nos parâmetros de configuração do algoritmo para minimizar as falsas suspeitas mantendo a detecção de falhas rápida.

Depois de processar os pacotes pendentes, se um *bag* não estiver vazio então é porque houve pacotes que foram encaminhados incorretamente para o servidor (linha 30). Por

Algorithm 1: Processamento de um pacote numa réplica i do BC.

when a replica (re)starts
begin
  forall the $forw = 0, ..., n-1$ **do**
    $Pend[forw] \leftarrow \emptyset$; $susp[forw] \leftarrow 0$; $asusp[forw] \leftarrow 0$
    $faulty[forw] \leftarrow false$; $ignore[forw] \leftarrow$ `IGNORE`
  send$(controller, \langle hello, i \rangle)$
when a packet r is received
begin
  $forw \leftarrow \mathsf{H}(r.src)\%n$
  while $faulty[forw]$ **do** $forw \leftarrow (forw+1)\%n$
  $Watchers \leftarrow \{(forw+j)\%n : j = 1, ..., 2f\}$
  if $i \in Watchers \cup \{forw\}$ **then**
    $dest \leftarrow$ getDestination(r)
    if $dest = \perp$ **then return**
    if $i = forw$ **then**
      send$(dest, \langle r, forw \rangle)$
    else
      $Pend[forw] \leftarrow Pend[forw] \cup \{\langle dest, \mathcal{T}_{now}, r \rangle\}$
when a request bag B^{dest}_{forw} is received
begin
  $suspect \leftarrow false$
  forall the $\langle dest, t, r \rangle \in Pend[forw]$ **do**
    if $r \in B^{dest}_{forw}$ **then**
      $Pend[forw] \leftarrow Pend[forw] / \{\langle dest, t, r \rangle\}$
      $B^{dest}_{forw} \leftarrow B^{dest}_{forw} / \{r\}$
    else if $t +$ `TIMEOUT` $< \mathcal{T}_{now}$ **then**
      $suspect \leftarrow true$
      send$(dest, \langle r, -1 \rangle)$
      $Pend[forw] \leftarrow Pend[forw] / \{\langle dest, t, r \rangle\}$
  if $B^{dest}_{forw} \neq \emptyset$ **then** $suspect \leftarrow true$
  if $ignore[forw] > 0$ **then** $ignore[forw]--$
  else if $faulty[forw] = false$ **then**
    if $suspect$ **then**
      $susp[forw]++$; $asusp[forw]++$
    else if $asusp[forw] > 0$ **then**
      $asusp[forw]--$
    if $(susp[forw] \geq$ `TH_SUSP`$) \vee (asusp[forw] \geq$ `TH_ASUSP`$)$ **then**
      send$(controller, \langle suspect, forw \rangle)$
when a packet $\langle alive, forw \rangle$ is received
begin
  $Pend[forw] \leftarrow \emptyset$; $susp[forw] \leftarrow 0$; $asusp[forw] \leftarrow 0$
  $faulty[forw] \leftarrow false$; $ignore[forw] \leftarrow$ `IGNORE`
when a packet $\langle faulty, forw \rangle$ is received
begin
  $faulty[forw] \leftarrow true$; $ignore[forw] \leftarrow$ `IGNORE`
when a packet $\langle update, new\_policy \rangle$ is received
begin
  updatePolicy(new_policy); $ignore[forw] \leftarrow$ `IGNORE`

exemplo, devido a alguma falta, o responsável pode ter modificado o conteúdo do pacote, e consequentemente, não foi encontrado o pacote correspondente na tabela de pendentes. Quando isto acontece o responsável é também considerado suspeito.

Se existirem rondas que ainda devem ser ignoradas, o contador de rondas a ignorar é decrementado (linha 31). Os vigias determinam se é necessário atualizar os contadores de suspeição. Um dos contadores ($susp$) nunca é decrementado para assegurar que uma réplica com faltas intermitentes será algures no tempo reiniciada. O outro contador é modificado de acordo com as ações da réplica responsável no último *bag* recebido (linhas 33-36). Por último, o vigia verifica se deve ser enviada uma mensagem de suspeição para o controlador, a indicar que a réplica deve ser reiniciada. São usadas duas constantes para verificar esta condição. O limite `TH_SUSP` define o número máximo de rondas em que ações de encaminhamento erradas podem ser observadas durante o tempo de vida de uma réplica. O `TH_ASUSP` especifica um limite semelhante, mas neste caso é o limite de rondas consecutivas em que uma réplica realizou ações erradas.

A réplica é informada pelo controlador quando outra réplica é recuperada ou é considerada incorreta (linhas 39 e 43). No caso de uma recuperação, a réplica reinicializa todas as suas tabelas. Adicionalmente, a política de distribuição usada (linha 16) pode ser atualizada pelo controlador através de uma mensagem de atualização, o que faz com que as réplicas invoquem as função *updatePolicy()* (linhas 46-48).

3.5.2 Servidor

Quando um servidor é iniciado, os *bags* que vão guardar a informação sobre os pacotes recebidos são limpos (ver algoritmo 2). É também inicializado um temporizador $tbag$, que irá expirar após `ROUND` do instante atual (linhas 1-5). É de salientar que o algoritmo usa dois *bags* para cada responsável: B e PB (*bag* da ronda anterior). A ideia é enviar os pacotes na ronda seguinte em que foram recebidos, para assegurar que as réplicas do BC não suspeitam de uma réplica correta porque os pacotes foram encaminhados próximos do final de uma ronda.

Relembrando, o `ROUND` é uma constante que define o intervalo durante o qual os servidores acumulam pacotes, antes de enviar a informação para os vigias. Consequentemente, este valor envolve algumas trocas. Intervalos maiores podem potencialmente fazer com que exista menos tráfego na rede mas faz com que a detecção de faltas demore mais tempo. Por outro lado, intervalos mais pequenos aumentam a carga da rede e os custos de processamento. No protótipo utilizamos $\texttt{ROUND} = 1$ segundo.

O servidor guarda os pacotes que recebe num *bag* associado ao responsável (linhas 6-9). Se o pacote tiver sido retransmitido então nenhuma informação é guardada porque o pacote foi enviado por um vigia (linha 28 do algoritmo 1). Quando o temporizador expira, o servidor verifica quais os vigias para cada responsável e envia o *bag* correspondente à última ronda (linhas 10-15). Por fim, o servidor guarda os *bags* atuais e reinicializa o

Algorithm 2: Processamento de um pacote no servidor i.

```
1  when a server starts
2  begin
3      forall the forw ∈ forwaders do
4          B^i_forw ← ∅; PB^i_forw ← ∅
5      timer(tbag, T_now + ROUND)
6  when a message ⟨r, forw⟩ is received
7  begin
8      if forw ≠ −1 then
9          B^i_forw ← B^i_forw ∪ {r}
10 when tbag expires
11 begin
12     forall the forw ∈ forwaders do
13         Watchers ← {(forw + j)%n : j = 1, ..., 2f}
14         forall the watcher ∈ Watchers do
15             send(watcher, ⟨PB^i_forw⟩)
16         PB^i_forw ← B^i_forw; B^i_forw ← ∅
17     timer(tbag, T_now + ROUND)
```

temporizador para mais uma ronda.

3.5.3 Controlador

O controlador é usado para reconfigurar o conjunto de réplicas, adicionando ou removendo BCs ao sistema. Adicionalmente, pode suportar a reconfiguração de políticas, por exemplo, através da atualização dos pesos de cada servidor (política *dynamic weighted round-robin* [21, 30]); adição ou remoção de servidores; ou adição de regras de lista negra ou outras definições para a *firewall* (por exemplo, o número máximo de pacotes que um cliente pode enviar por segundo). O controlador atualiza o sistema alterando as definições de distribuição de tráfego do *switch* e informa as réplicas das modificações.

O protocolo de atualização executado pelo controlador segue os seguintes passos: *(1)* o controlador remove do *switch* todas as regras de encaminhamento de tráfego para as réplicas do BC; *(2)* as réplicas são notificadas sobre as atualizações; e *(3)* as regras de encaminhamento, que agora contêm o novo conjunto de réplicas, são adicionadas ao *switch*. O primeiro passo deste protocolo é opcional – o desenho inexato permite que o sistema continue a executar com configurações parcialmente incorretas durante o período de atualização.

Para evitar problemas de ponto único de falha, o controlador pode ser replicado usando o Paxos tal como é feito nas infraestruturas da Microsoft e da Google [48, 26].

3.6 Discussão

3.6.1 Representação Compacta dos *Bags*

Os *bags* usados pelos servidores podem potencialmente crescer até tamanhos muito grandes, se tiverem que acomodar mais informação sobre os pacotes encaminhados. Além disso, uma vez que os pacotes podem ser corrompidos arbitrariamente pelos BCs incorretos, os dados armazenados necessitam de contemplar todo o conteúdo do pacote. Numa implementação ingénua, isto pode trazer elevados custos uma vez que os *bags* têm que ser enviados para os vigias.

Para resolver este problema, é necessário aproximar o comportamento dos *bags* do comportamento de um *bloom filter* [17]. Cada vez que um pacote chega ao servidor é calculada uma *hash* do pacote que serve como *input* para o *bloom filter*. Quando o temporizador expira, o servidor envia o conteúdo do *bloom filter* para os vigias. Do lado do vigia, para verificar se um pacote foi encaminhado para o servidor, é computada a *hash* desse pacote. Depois, o *bloom filter* é questionado se a *hash* pertence ao seu conteúdo. Para assegurar uma taxa de falsos positivos mais baixa é necessário usar uma função de *hash* com uma boa capacidade de resistência a colisões (por exemplo, SHA-1).

Os *bloom filters* têm a característica de nunca produzirem falsos negativos. Assim, um BC nunca será suspeito de ter corrompido um pacote. Por outro lado, podem gerar falsos positivos (com uma pequena probabilidade), i.e., os *bloom filters* podem retornar que uma *hash* pertence ao seu conteúdo quando na realidade não pertence. Quando esta situação acontece, um BC pode considerar que um pacote foi corretamente encaminhado, quando na verdade foi perdido. Este problema não é grave para a nossa proposta já que: a aplicação deve ser capaz de retransmitir o pacote; e o BC incorreto vai continuar a descartar pacotes, e mais tarde será considerado incorreto.

3.6.2 Problemas de Assincronia

A assincronia da rede e dos componentes do sistema pode introduzir diferentes atrasos na entrega dos pacotes. Um dos casos extremos ocorre quando um servidor recebe um pacote, adiciona-o ao *bag*, e envia o *bag* para o vigia antes do vigia receber o pacote. Quando isto acontece, o vigia considera que o responsável é incorreto, uma vez que o responsável alegadamente criou ou corrompeu pacotes. Outro caso extremo ocorre quando um responsável está atrasado. Neste caso, os vigias recebem o pacote e um ou mais *bags* do servidor, que não contêm o pacote recebido. Algures no tempo, os vigias vão considerar que o responsável não enviou o pacote e detectar a réplica como incorreta.

Não é expectável que estes cenários sejam verificados com frequência, e tipicamente violam as assunções do nosso modelo. Um sistema configurado de forma apropriada usa valores para os tempos transmissão e processamento máximos corretos, fazendo com que os temporizadores sejam inicializados corretamente. É também possível que exista

um processo em segundo plano que verifica as condições da rede e ajusta os valores do TIMEOUT e do ROUND para que tenham em conta os atrasos da rede. Apesar disso, os protocolos prevêem que alguns destes erros possam existir. O contador $asusp$ é decrementado assim que o sistema volta a operar de forma correta. O contador $susp$ vai continuar com os valores das suspeições mas estes têm um limite muito superior para minimizar as falsas suspeitas.

O mecanismo de recuperação pode também ser afetado pela assincronia do sistema. Alguns BCs podem considerar que uma réplica é incorreta, enquanto outras ainda a consideram correta. Quando isto acontece, alguns pacotes vão ser processados de forma incoerente pelas réplicas do BC, por exemplo, diferentes réplicas escolhem diferentes servidores de destino (linha 13 do algoritmo 1). No entanto, o sistema foi desenhado para lidar com este tipo de incoerências. As réplicas incorretas são algures no tempo identificadas por todos os vigias, e o controlador pode ser configurado para ignorar as suspeições durante algumas rondas. Além disso, se existir uma falsa suspeita, o contador $asusp$ vai sendo decrementado enquanto o sistema retoma o estado normal.

3.6.3 Correção das Propriedades

O sistema satisfaz as duas propriedades introduzidas na secção 3.3.3. A primeira propriedade declara que se um pacote for recebido por pelo menos $f + 1$ vigias corretos, será encaminhado para o servidor final. Isto acontece devido ao mecanismo de temporização usado pelos vigias: se um pacote ficar na tabela $Pend[forw]$ mais do que TIMEOUT, o vigia retransmite o pacote para o servidor final (linhas 26-29 do algoritmo 1).

A segunda propriedade está relacionada com a detecção de faltas e isolamento, e define que uma réplica incorreta será removida do sistema algures no tempo. Esta propriedade é satisfeita através do uso dos contadores $susp$ e $asusp$. Uma réplica que se comporta de forma incorreta durante pelo menos TH_SUSP rondas ou processa incorretamente os pacotes durante TH_ASUSP rondas consecutivas, é considerada incorreta pelos vigias (linhas 37-38 do algoritmo 1). Quando o controlador recebe $f + 1$ votos de suspeição, atualiza o sistema indicando que uma réplica é incorreta (linhas 43-45 do algoritmo 1) e modifica as regras no *switch* para que os pacotes não sejam mais disseminados para a réplica incorreta. É de salientar que uma vez que são necessários $f+1$ votos para remover uma réplica (garantido que pelo menos 1 réplica correta suspeitou), significa que são necessários $2f$ vigias para cada responsável.

3.6.4 Comportamentos Bizantinos

No nosso modelo de faltas assumimos que o BC pode ter comportamentos bizantinos, mais concretamente, assumimos que podem ter comportamentos bizantinos sistemáticos. O sistema foi desenhado para tolerar f faltas com $2f + 1 + k$ réplicas do BC. Os cenários

previstos em que uma réplica sofre uma falta forma bizantina são os seguintes:

Cenário 1: uma réplica incorreta não encaminha os pacotes para o servidor. Este tipo de faltas são ditas "por omissão", e são equivalentes às faltas por paragem. Estas faltas são detectadas recorrendo ao temporizador `TIMEOUT` utilizado para cada pacote: quando este temporizador expira, a réplica responsável é considerada suspeita e os contadores são incrementados.

Cenário 2: uma réplica cria e encaminha mensagens que não foram enviadas por um cliente. Neste cenário, na ronda seguinte o *bag* recebido pelos vigias contém pacotes a mais e a réplica responsável é considerada suspeita.

Cenário 3: uma réplica corrompe e encaminha um pacote do cliente. A réplica incorreta é detectada por duas razões: *(1)* existe um pacote que não era suposto no *bag* da ronda seguinte; e *(2)* o temporizador do pacote que deveria ter sido encaminhado expira.

Cenário 4: uma réplica encaminha as mensagens para um servidor errado. A réplica incorreta é detectada pelas mesmas razões do cenário 3: *(1)* é recebido um *bag* com pacotes, e o *bag* deveria estar vazio; e *(2)* o temporizador do pacote que deveria ter sido encaminhado expira.

Cenário 5: uma réplica envia um voto errado para o controlador. Mesmo que uma réplica incorreta envie um voto de suspeição, enquanto o controlador não receber $f + 1$ votos, nenhuma ação é tomada, e a réplica correta não é removida. No entanto, neste cenário existe uma limitação, o algoritmo não detecta que a réplica vigia é incorreta.

3.7 Sumário

Neste capítulo motivámos a necessidade de um novo modelo de faltas, especificámos o nosso desenho inexato e apresentámos o modelo do nosso sistema. De seguida descrevemos os procedimentos do algoritmo de tolerância a faltas bizantinas sistemáticas e finalizámos com uma discussão dos potenciais problemas do desenho inexato e uma descrição do tratamento de vários cenários de faltas. No próximo capítulo apresentamos os detalhes da implementação deste algoritmo.

Capítulo 4

Implementação

Este capítulo apresenta os detalhes de implementação do BC tolerante a faltas bizantinas sistemáticas. Começamos por descrever as arquiteturas usadas nos sistemas atuais, e de seguida guiamos o leitor pelo processo de desenvolvimento do protótipo, explicando os diversos desafios e as soluções encontradas.

4.1 Arquitetura Tolerante a Faltas por Paragem

Os mecanismos para tolerância a faltas por paragem são mais simples que os mecanismos para tolerância a faltas bizantinas. A solução mais comum consiste em ter uma réplica mestre do BC que encaminha todos os pacotes dos clientes para os servidores, e uma ou mais réplicas secundárias do BC que substituem o mestre no caso de falta (ver figura 4.1). Neste tipo de sistemas o protocolo de tolerância a faltas tem de detectar que a réplica mestre deixou de responder, removê-la do sistema e eleger uma réplica secundária como mestre. O mecanismo típico de detecção de faltas utiliza *heartbeats*, i.e., a réplica mestre envia uma mensagem periódica para avisar as réplicas secundárias de que está ativa. Se durante um período de tempo as réplicas secundárias não receberem o *heartbeat*, consideram que o BC mestre falhou.

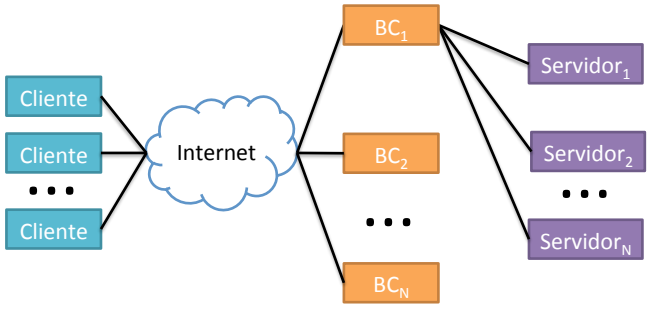

Figura 4.1: Sistema com vários BCs e vários servidores.

Esta é a solução de tolerância a faltas tipicamente usada em sistemas com balanceadores de carga.

4.2 Arquiteturas de Balanceamento de Carga para Disseminação de Pacotes

Nesta secção apresentamos várias arquiteturas que permitem que todas as réplicas do BC recebam o mesmo pacote. A solução mais simples para atingir este objetivo é fazer com que a aplicação do cliente envie os pacotes para todas as réplicas. Esta abordagem não foi explorada porque queremos que o sistema seja transparente para o cliente, i.e., que o cliente interaja com o nosso sistema da mesma forma que interage com um sistema não replicado.

4.2.1 *HUB*

O *HUB* é um dispositivo com múltiplas interfaces de comunicação, responsável por replicar os dados que entram por todas as suas interfaces. A figura 4.2 mostra como o *HUB* pode ser usado no nosso sistema. O *HUB* é mais lento que os *switches* mais recentes e a taxa de colisão de pacotes é maior[1]. Os *HUBs* mais recentes chegam a atingir taxas de transmissão de 100 Mbit/s, enquanto que os *switches* mais recentes já atingem taxas de 100 Gbit/s.

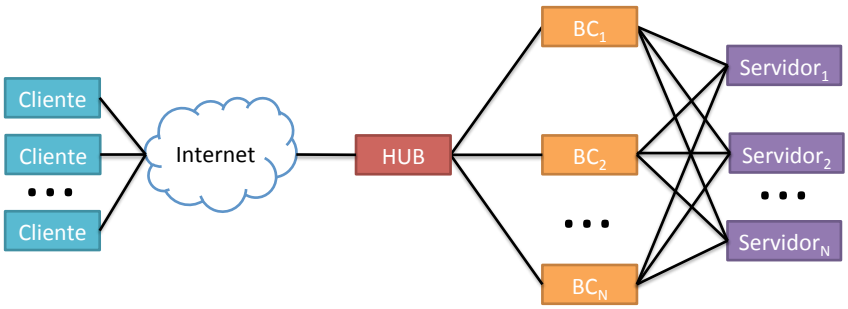

Figura 4.2: Sistema com vários BCs, vários servidores e um *HUB*.

Uma das exigências relativamente ao BC é que este seja eficiente. Por essa razão, não utilizamos um *HUB* no nosso sistema.

4.2.2 *Front-End*

Nesta arquitetura o *HUB* foi substituído por outro componente, denominado por *front-end*, que emula o comportamento do *HUB* (ver figura 4.3). O *front-end* consiste numa aplicação que envia em *unicast* os pacotes para todas réplicas do BC.

É possível também usar o *front-end* para encaminhar os pacotes dos clientes para um IP de *multicast* ou de *broadcast*. Estas soluções escalam melhor que a aplicação de *unicast* porque o custo é constante, enquanto que com *unicast* o custo cresce linearmente com o número de réplicas.

[1] http://www.ccontrols.com/pdf/Extv3n3.pdf

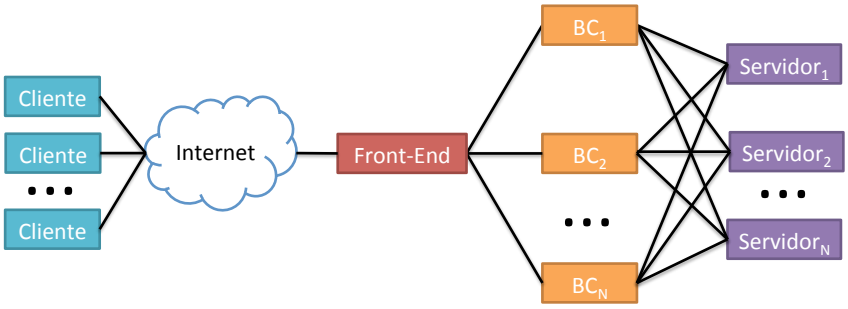

Figura 4.3: Sistema com vários BCs, vários servidores e um *front-end* para disseminação.

4.2.3 *Switch*

Esta arquitetura usa as funcionalidades *OpenFlow* [38] do *switch* para disseminar pacotes para todas as interfaces das réplicas do BC (ver figura 4.4). Desta forma conseguimos ter um *switch*, que tem um bom desempenho a disseminar os pacotes. É possível também usar as funcionalidades *OpenFlow* para adicionar a remover regras dinamicamente, isto permite ao algoritmo remover facilmente os componentes incorretos do sistema.

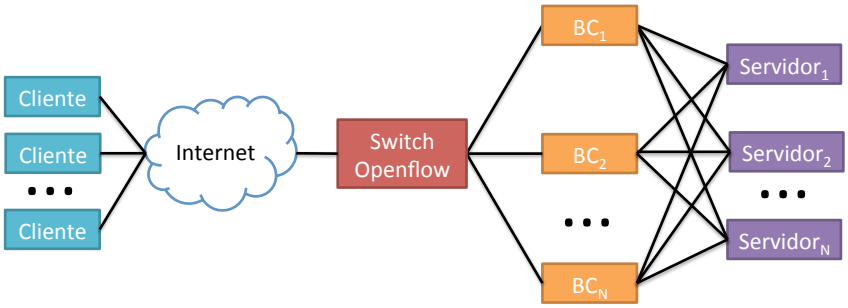

Figura 4.4: Sistema com vários BC, vários servidores e um *switch OpenFlow*.

4.2.4 Discussão das Alternativas e Implementações Testadas

A solução que obriga o cliente a disseminar os pacotes para as réplicas foi excluída porque queremos que a nossa solução seja transparente para o cliente. A segunda solução (utilizando o *HUB*) foi também descartada devido ao seu baixo desempenho.

Dentro das opções que fazem uso de um *front-end* para disseminar os pacotes, a opção que recorre a uma aplicação para enviar os pacotes em *unicast* é a menos escalável e portanto também não foi considerada como opção. As soluções de encaminhamento *multicast* ou *broadcast* foram testadas recorrendo ao uso do *iptables* [39] e à tabela *Address Resolution Protocol* (ARP)[2]. Foram introduzidas regras no *iptables* que forçam o encaminhamento dos pacotes (cujo destino é o endereço IP do serviço) para um endereço IP

[2]A tabela ARP é utilizada para traduzir endereços IP em endereços MAC.

do serviço. Na tabela ARP inserimos uma entrada que traduz este IP num endereço MAC de *multicast* ou *broadcast*, e assim o pacote é encaminhado para as réplicas do BC.

A arquitetura com o *switch* faz uso das funcionalidades *OpenFlow* que permitem a inserção de regras para replicar o tráfego por várias portas. Segundo as experiências descritas no capítulo 5 esta opção não tem um bom desempenho, mas esta limitação está relacionada com o modelo do *switch* usado nas experiências (ver explicação na secção 5.4.1).

É de salientar que para todas as alternativas discutidas (com exceção da que obriga o cliente a disseminar os pacotes) é necessário que todas as réplicas tenham mesmo endereço IP (o endereço do serviço) porque, embora os pacotes sejam replicados e encaminhados para as diferentes réplicas, o endereço IP de destino mantém-se.

Avaliando as várias opções, optámos pela solução que faz uso do *switch* por duas razões: *(1)* permite-nos usar um controlador *OpenFlow* no nosso algoritmo para facilitar a adição e remoção de componentes; e *(2)* usar uma solução com *front-end* implicaria assumir que este componente era confiável.

4.3 Balanceador de Carga Tolerante a Faltas Bizantinas Sistemáticas

A nossa implementação do balanceador de carga consiste em duas aplicações (também designadas por módulos), uma que é executada juntamente com as réplicas do BC e outra que é executada juntamente com os servidores. Em seguida, estas aplicações são descritas de uma forma cronológica: começamos por apresentar uma implementação inicial e descrevemos as alterações que fizemos até atingir a versão final.

4.3.1 Balanceador de Carga Inicial

Foram exploradas duas implementações de balanceamento de carga no nível aplicacional: *Apache httpd-bc* [11] e LVS [52]. O nosso algoritmo foi desenhado para fazer balanceamento de carga e tolerância a faltas ao nível aplicacional, e uma vez que estas implementações já forneciam tolerância a faltas por paragem, inicialmente o algoritmo seria integrado num destes BCs como uma melhoria ao mecanismo de tolerância a faltas.

O LVS é bastante complexo, o que dificulta a integração de um algoritmo de balanceamento de carga diferente dos fornecidos. Por outro lado, a implementação da *Apache* é mais simples e modular, o que facilita a integração de novos algoritmos. Existem dois módulos que implementam o BC no *httpd-bc*: o módulo *Proxy* [11] e o módulo *Connectors* [12].

Uma vez que as faltas podem ocorrer a mais baixo nível, foi necessário alterar o mecanismo de tolerância a faltas do algoritmo para o nível 4 (nível TCP/IP). Como descrito

no capítulo 3, é necessário que todos os BC recebam todos os pacotes, e para que isso seja possível usando o *httpd-bc*, é necessário uma ligação TCP/IP entre o cliente e todas as réplicas do BC. Para que não existam conflitos entre as réplicas é preciso interceptar, alterar e aceitar ou descartar os pacotes. Para além de tornar a solução mais complexa, tornava também o sistema mais lento, por isso, removemos o *httpd-bc* do sistema e implementámos um encaminhador de pacotes de raiz.

4.3.2 Módulo no Balanceador de Carga

Como todas as réplicas recebem os pacotes do cliente, todas as réplicas tentam estabelecer uma ligação com o cliente, mas uma ligação TCP só pode ser estabelecida entre duas entidades. Para isso, é necessário interceptar os pacotes antes de serem entregues à aplicação. A primeira implementação recorre ao *iptables* e à biblioteca *nfqueue* [40] que permitem interceptar os pacotes e adicioná-los a uma fila. É preciso também implementar um software que retire os pacotes da fila e os descarte ou entregue à aplicação, dependendo do papel de cada réplica. Este software foi implementado na linguagem de programação C e é o componente que vai executar o protocolo de tolerância a faltas. Com a implementação do algoritmo surgiram diversos desafios, nomeadamente:

Identificação de uma réplica do BC. O algoritmo pressupõe que os servidores conseguem detectar qual a réplica que encaminhou o pacote, e como inicialmente todas as réplicas precisavam de estar configuradas com o mesmo endereço IP e MAC, era impossível o servidor saber qual a réplica responsável por encaminhar o pacote. A primeira solução foi acrescentar o identificador da réplica ao pacote que é encaminhado para o servidor.

Quando uma réplica é a responsável, decide qual o servidor de destino do pacote através do algoritmo de balanceamento de carga, e encaminha o pacote. Para isso, altera o endereço IP e o porto de destino para o endereço IP e porto do servidor, acrescenta o seu identificador e recalcula os *checksums* IP e TCP do pacote (passo 1 da figura 4.5).

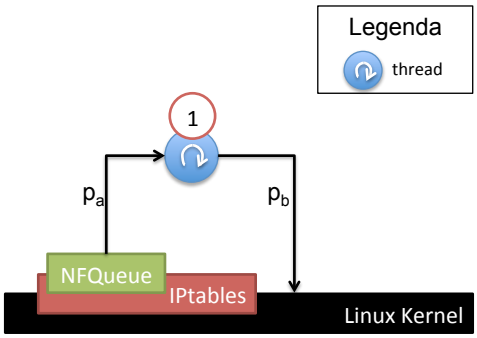

Figura 4.5: Implementação inicial do módulo no BC.

Ao usar um identificador único para cada réplica temos de lidar com duas limitações: *(1)* o identificador não pode ser alterado por uma réplica incorreta porque pode induzir as réplicas em erro; e *(2)* os pacotes não podem ter mais que 1496 bytes (é de relembrar que o tamanho máximo de um pacote IP é 1500 bytes) porque não é possível acrescentar os 4 bytes do identificador sem fragmentar o pacote. Como numa implementação posterior deixa de ser necessário interceptar os pacotes e apenas é necessário escutá-los (porque removemos o *httpd-bc* do nosso sistema), as réplicas do BC não precisam de ter o mesmo endereço IP e MAC. Desta forma, é possível remover o identificador do pacote e as réplicas passam a ser identificadas pelo seu endereço MAC. Para escutar os pacotes recorremos à biblioteca *PCAP* ao invés da biblioteca *nfqueue*. Com o *PCAP* as réplicas escutam os pacotes enviados pelos clientes para o endereço IP do serviço e descartam-nos em seguida (passo 1 da figura 4.6).

Perda de pacotes nos vigias. Na implementação inicial as réplicas vigias perdiam mais de 10% dos pacotes. Quantos mais pacotes enviados, maior a percentagem de pacotes processadores pela réplica responsável e menor o número de pacotes que chegavam aos vigias. A verificação dos *bags* era feita por uma *thread* dedicada (passo 5 da figura 4.6), e as perdas podiam ser o resultado de problemas de concorrência. Depois de vários testes, chegámos à conclusão de que a perda de pacotes era o resultado do uso de uma lista simples para guardar os pacotes encaminhados pelos responsáveis. Alterar a implementação para uma lista duplamente ligada reduziu as perdas para menos de 0.1%.

Indexação. A lista de pacotes guardada pelos vigias começou por ser indexada apenas pelo responsável, i.e., cada vigia guardava apenas uma lista para cada responsável. Para melhorar o desempenho, cada vigia passou a ter $|\mathcal{S}|$ listas de cada responsável (relembramos que $\mathcal{S}$ é o conjunto de servidores). Desta forma, quando um vigia recebe um *bag* enviado pelo servidor s com os pacotes encaminhados por r, apenas precisa de verificar os pacotes encaminhados por r para s em vez de verificar todos os pacotes encaminhados por r.

***Threads*.** A implementação foi desenvolvida para tirar partido das máquinas com vários processadores e faz uso de *threads* para executar as diferentes tarefas do algoritmo. Existem quatro tipos de *threads*: *(1)* uma *thread* que recebe os pacotes do cliente e coloca-os numa fila (passo 2 da figura 4.6); *(2)* uma *thread* que recebe os *bags* e verifica as faltas das outras réplicas (passo 5 da figura 4.6); *(3)* um conjunto de $|\mathcal{S}|$ *threads* que processam os pacotes recebidos pela primeira *thread* de acordo com o papel da réplica (passos 3_a e 3_b da figura 4.6); *(4)* uma *thread* que retira os pacotes da última fila e encaminha-os para o servidor de destino (passo 4 da figura 4.6).

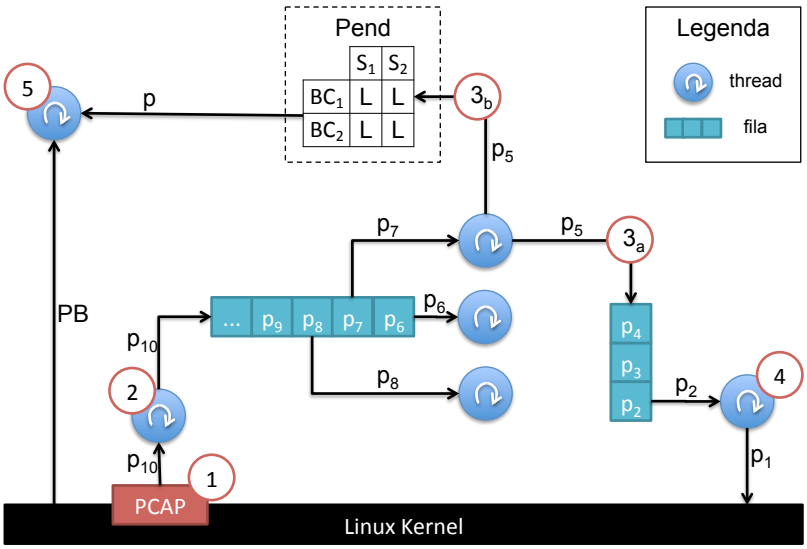

Figura 4.6: Implementação final do módulo no BC.

4.3.3 Módulo no Servidor

No servidor existe um módulo que, numa primeira implementação, intercepta os pacotes antes de serem entregues ao servidor Web, usando a biblioteca *nfqueue*. Ao retirar um pacote da fila, o módulo remove e verifica o identificador da réplica e: *(1)* coloca o pacote recebido no *bag* correspondente; *(2)* entrega o pacote ao servidor Web (passo 1 da figura 4.7). O servidor Web recebe o pacote e responde ao cliente (passo 2 da figura 4.7). Novamente através da biblioteca *nfqueue*, a resposta é interceptada e o módulo servidor altera o endereço IP e o porto de origem para o endereço IP e porto do serviço (passo 3 da figura 4.7).

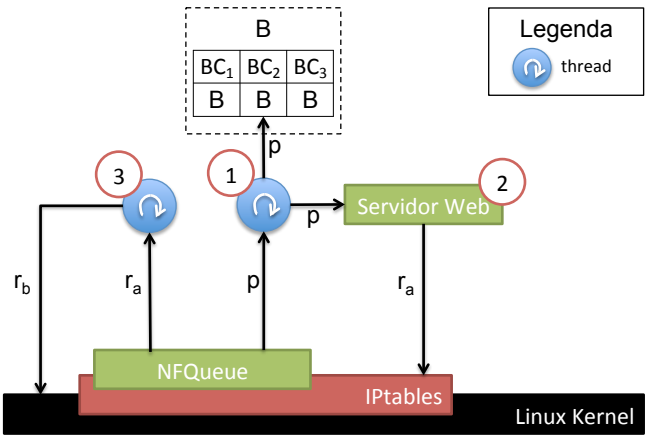

Figura 4.7: Implementação inicial do módulo no servidor.

Depois de alguns testes em que os resultados não foram satisfatórios, foi necessário melhorar o desempenho deste módulo. Para isso, foi removida a segunda intercepção de pacotes (passo 3 da figura 4.7). Isto foi possível através da adição de uma interface virtual

que escuta o endereço IP do serviço, permitindo que o servidor Web responda diretamente ao cliente com esse IP.

Identificação de uma réplica do BC. Tal como explicado na secção 4.3.2, é possível remover o identificador que é acrescentado pelas réplicas aos pacotes. Desta forma, é possível (e necessário para a identificação das réplicas) substituir a biblioteca *nfqueue* pela biblioteca *PCAP*. Os pacotes deixam de ser interceptados e passam a ser apenas escutados (passo 1 da figura 4.8). Como a biblioteca *PCAP* fornece os cabeçalhos *ethernet* é possível identificar as réplicas pelo seu endereço MAC.

***Threads*.** Para resolver o problema de existirem falsos positivos (ver secção 3.5.2), adicionámos à implementação deste módulo a segunda tabela com os *bags* da ronda anterior. Existem duas *threads* neste módulo: *(1)* uma que recebe os pacotes da biblioteca *PCAP* e os coloca nos *bags* da ronda atual (passo 2 da figura 4.8); e *(2)* uma que envia os *bags* da ronda anterior para os vigias (passo 3 da figura 4.8) e em seguida atualiza as tabelas dos *bags* (passo 4 da figura 4.8).

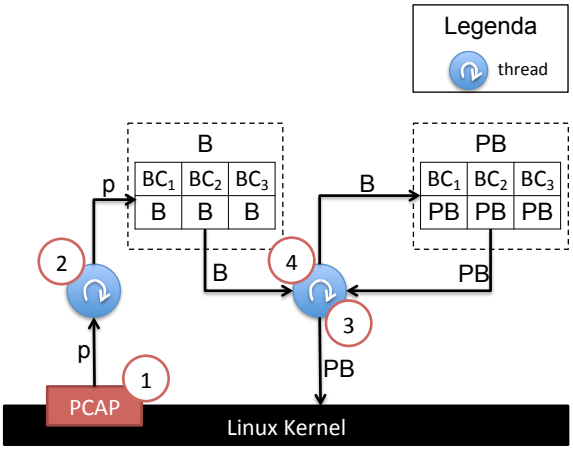

Figura 4.8: Implementação final do módulo no servidor.

4.3.4 Implementação do Controlador

A implementação do controlador é apenas uma prova de conceito usada para testar o algoritmo. Tal como na implementação dos módulos, o controlador utiliza a biblioteca *PCAP* para escutar as mensagens enviadas pelas réplicas do BC e pelos servidores (passo 1 da figura 4.9). Sempre que um dos componentes tem que comunicar com o controlador (por exemplo, quando inicia e quer "entrar" no sistema) envia uma mensagem através de *User Datagram Protocol* (UDP) para o endereço IP do controlador. O controlador ao receber a mensagem do *PCAP* verifica qual é o tipo da mensagem (passo 2 da figura 4.9).

Existem dois tipos de mensagens: *(1)* mensagens de inicialização / recuperação; e *(2)* mensagens de suspeição.

Se uma mensagem é do tipo *(1)*, o controlador atualiza a lista de componentes do sistema (vista) adicionando o novo componente (passo 3 da figura 4.9) – se o novo componente é uma réplica do BC, o controlador atualiza também as regras do *switch* (passo 5 da figura 4.9). No final, envia uma mensagem para todos os componentes do sistema a informar sobre o novo componente. Se uma mensagem é do tipo *(2)*, o controlador atualiza a tabela de votos (passo 4 da figura 4.9) – se o número de votos é igual a $f + 1$, o controlador atualiza a lista de componentes, altera as regras do *switch* e envia a nova lista de componentes (sem a réplica incorreta) para os componentes do sistema.

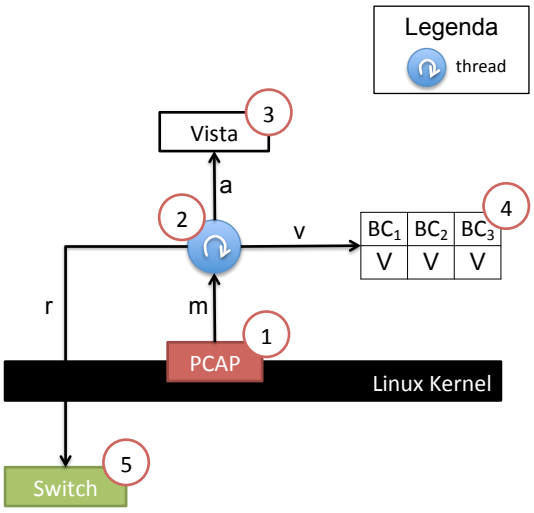

Figura 4.9: Implementação do controlador.

4.4 Protótipo Final

A figura 4.10 mostra todos os passos do algoritmo nos dois módulos implementados com todas as otimizações descritas nas secções anteriores. O cliente envia um pedido para o sistema que é escutado pelo módulo no BC usando a biblioteca *PCAP* (passo 1). A primeira *thread* deste módulo recebe o pacote da biblioteca *PCAP* e coloca-o numa fila (passo 2). Existem $|\mathcal{S}|$ *threads* que retiram e processam os pacotes da fila, calculando o servidor para onde encaminhar e colocando os pacotes noutra fila (passo 3_a), ou colocando-os numa lista caso sejam vigias (passo 3_b da figura 4.10). A última *thread* deste módulo retira os pacotes da fila e encaminha-os para os servidores selecionados (passo 4).

O módulo no servidor recebe um pacote da biblioteca *PCAP* (passo 5 da figura 4.10), e coloca-o no *bag* correspondente à réplica que encaminhou o pacote (passo 6). Outra *thread* envia os *bags* da ronda anterior para os vigias (passo 7), coloca os *bags* da ronda atual na tabela de *bags* da ronda anterior, limpando os primeiros (passo 8).

Quando a *thread* do módulo no BC receber o *bag* verifica se não existem réplicas incorretas no sistema (passo 9).

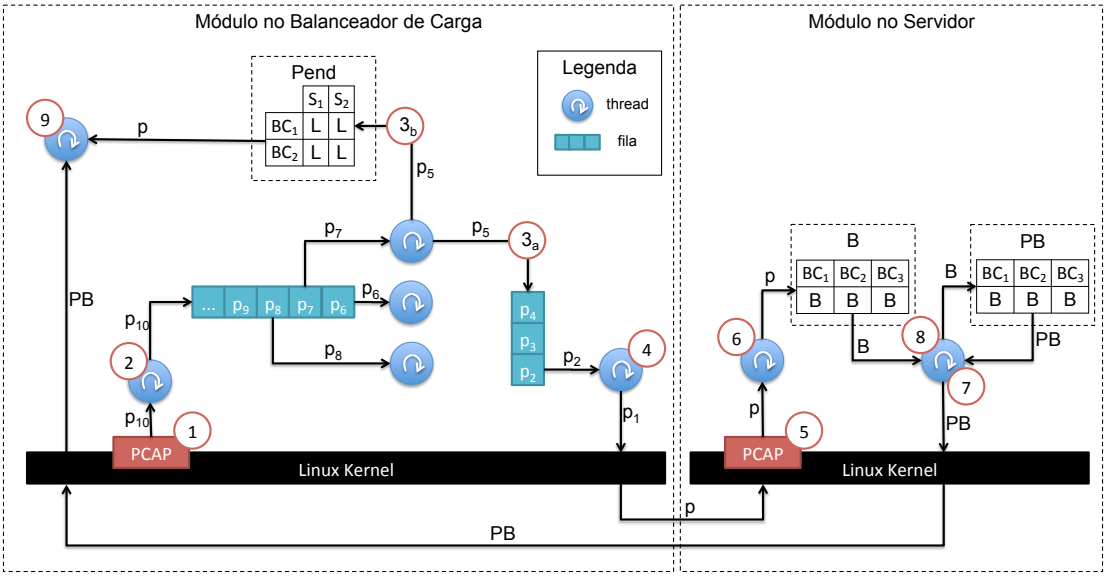

Figura 4.10: Implementação final dos módulos no BC e no servidor.

Os principais desafios da implementação foram descobrir a melhor arquitetura para implementar o BC e resolver os problemas do algoritmo, descobertos durante a fase de implementação. O resultado final são dois módulos: um no BC com 11 ficheiros que perfazem um total de 1637 linhas de código; e outro no servidor com 9 ficheiros que perfazem um total de 938 linhas de código.

4.5 Sumário

Neste capítulo discutimos diversas arquiteturas e técnicas de disseminação de pacotes para várias réplicas. Depois apresentámos os passos seguidos até à implementação final do protótipo e descrevemos as diversas opções tomadas. No final, apresentámos o protótipo de um balanceador de carga ao nível do utilizador que usa bibliotecas livres que interagem de forma eficiente com as diversas camadas da rede. No capítulo seguinte apresentamos várias experiências de desempenho e tolerância a faltas para avaliar o nosso protótipo.

Capítulo 5

Avaliação e Resultados

Neste capítulo descrevemos as experiências realizadas para avaliar o nosso protótipo analisando o seu desempenho, consumo de CPU, tolerância a faltas e escalabilidade. Comparamos também o desempenho com outros BCs usados atualmente e descrevemos os componentes utilizados no protótipo, a metodologia seguida nas experiências e apresentamos os resultados obtidos.

5.1 Metodologia

A avaliação tem por base um conjunto de servidores HTTP, mais concretamente servidores *httpd* da Apache [4], que podem ser acedidos através de um único endereço IP do serviço. O protótipo usa uma política de distribuição simples, cuja seleção do servidor final é determinada pela origem do pacote (endereço IP e porto de origem do cliente) e pelo número total de servidores[1].

Realizámos dois tipos de experiências: *1)* as que mostram o desempenho e escalabilidade do protótipo; e *2)* as que mostram o comportamento do protótipo nos cenários de faltas. Para *1)* apresentamos a média de pedidos que o servidor responde por segundo, e para *2)* mostramos os instantes de detecção e remoção da réplica incorreta. Todas as experiências foram repetidas um mínimo de 5 vezes, e não reportamos os desvios padrão porque apresentam valores abaixo dos 5% em todas as experiências. Nas experiências consideramos que uma ronda demora um segundo ($\texttt{ROUND} = 1$).

Nas experiências de desempenho as capacidades de *firewall* do BC foram desativadas para que a comparação entre BCs seja mais justa e nas experiências com cenários de falta foi considerado que apenas uma réplica era incorreta ($f = 1$).

Benchmark. Como cliente/gerador de pedidos, usámos o ApacheBenchmark [5] e não foi utilizada nenhuma aplicação específica nos servidores, além do servidor Web da *Apa-*

[1]Mais especificamente, $servidor = Hash(pedido.origem)\%N_{servidores}$. Esta política pode ser alterada para considerar pesos para cada servidor

che. Esta ferramenta permite especificar o número de pedidos concorrentes, o número total de pedidos e o tempo de execução de cada experiência. Permite também fazer pedidos POST, método que suporta a realização de pedidos com mais de 8 Kbytes (tamanho máximo de um pedido GET). O número de pedidos concorrentes pode também ser considerado como o número de clientes concorrentes a aceder ao servidor. Relembramos que, tal como já mencionado em capítulos anteriores, um pedido é equivalente a um pacote, com exceção de pedidos cujo tamanho ultrapasse os 1500 bytes.

Bloom Filter. Aceitando 1% de falsos positivos, o tamanho do *bloom filter* usado nas experiências é 99846 bytes. Os *bloom filters* são enviados para os vigias em cada ronda através da mesma rede em que os pacotes são encaminhados. O seu tamanho reduzido traduz-se em apenas 0.08% da utilização da rede, um valor praticamente nulo considerando o tráfego gerado nas experiências.

5.2 Objectivos

O objetivo destas experiências foi responder a quatro questões principais:

- *Desempenho:* Quantos pacotes por segundo consegue o protótipo distribuir? Qual é o desempenho do protótipo quando comparado com outros BCs, como o LVS [52] e o *httpd-bc* da Apache (*httpd-bc*) [11]? Qual o desempenho do protótipo ao variar o tamanho dos pedidos e das respostas?
- *Escalabilidade:* Como se comporta o sistema quando são adicionados mais servidores ou réplicas do BC?
- *Confiabilidade:* Qual o impacto na latência do serviço? Como é que a variação dos diferentes parâmetros do algoritmo afeta a detecção e remoção de faltas no protótipo? Como se comporta o protótipo em diversos cenários de faltas sistemáticas?
- *Impacto do desenho inexato:* Qual é o impacto de atualizar a política de distribuição (que pode fazer com que diferentes réplicas tomem diferentes decisões)?

5.3 Configuração

Nesta secção descrevemos o *hardware* usado no protótipo e a figura 5.1 mostra a arquitetura de rede utilizada nas experiências. Todos os componentes do sistema foram executados em servidores *Dell PowerEdge R410*[2]. Cada máquina tem um processador *Intel Xeon E5520*[3] com dois CPU com quatro *cores* cada. Cada *core* consegue executar

[2]http://www.dell.com/pt/business/p/poweredge-r410/pd
[3]http://ark.intel.com/Product.aspx?id=40200&code=Xeon+E5520

duas *threads* nativas em simultâneo. Cada processador tem uma velocidade de relógio de 2.27 *GHz*, 1 *MB* de memória *cache* L2 e 8 *MB* de memória *cache* L3. As máquinas têm 32 *GB* de memória *DDR3* que atingem uma velocidade de 1066 *MHz* e uma interface de rede *Broadcom NetXtreme II BCM5716 Gigabit Ethernet*.

Para fazer a comunicação entre as máquinas o protótipo usa um *switch HP Procurve 3500 yl*[4] de 24 portas que suporta a tecnologia *OpenFlow* [38] e atinge velocidades de transferência de 1 *Gbit/s*.

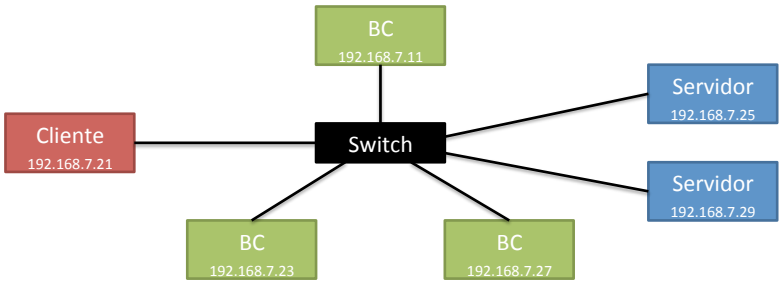

Figura 5.1: Arquitetura da rede do protótipo usada nas experiências.

5.4 Desempenho

Nesta secção comparamos as técnicas de disseminação e captura de pacotes, apresentamos os resultados de desempenho e comparamos o nosso protótipo com outros BCs.

5.4.1 Disseminação de Pacotes

Começamos por discutir as diferentes técnicas que permitem a disseminação do tráfego para as várias réplicas do BC e as limitações do nosso hardware. A figura 5.2 mostra a transmissão máxima de pacotes utilizando diferentes técnicas. Estas experiências fazem com que os clientes inundem completamente a rede com pacotes grandes (1500 bytes) e pequenos (185 bytes).

Como mostra a figura, o uso de regras *OpenFlow* multiportas reduz drasticamente o desempenho do sistema. Isto deve-se ao facto do *switch* avaliar as regras multiportas por software, impondo uma limitação de 10000 pacotes por segundo (incluindo pacotes de controlo do sistema) [2], o que se traduz em pouco mais de 3000 pedidos HTTP processados por segundo. Se usarmos um componente específico para fazer *broadcast* dos pacotes para todas as réplicas do BC, o desempenho varia entre os 9000 pacotes grandes e 60000 pacotes pequenos por segundo. Esta solução não foi escolhida, como já foi mencionado, para evitar o uso de mais um componente confiável na arquitetura.

Como base de comparação, apresentamos também o desempenho do sistema ao usar regras *OpenFlow* de porta única, em que os pacotes são processados por um *hardware*

[4]http://h30094.www3.hp.com/product.aspx?sku=10232286

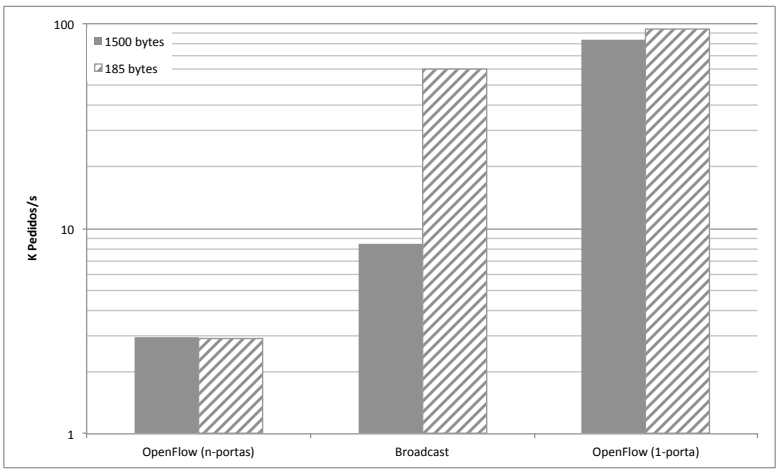

Figura 5.2: Desempenho do switch utilizando diferentes técnicas de disseminação (eixo y em escala logarítmica).

dedicado. Com esta regra e considerando os pedidos com 1500 bytes, o *switch* processa 30 vezes mais pacotes por segundo e utiliza a taxa de transmissão máxima da rede (1 Gbit/s). É de salientar que a diferença de tamanho dos pedidos tem pouco impacto no desempenho do *switch*.

A limitação de desempenho das regras multiportas – que se deve particularmente ao dispositivo que usámos (um dos primeiros da HP a suportar a tecnologia *OpenFlow*) – força-nos a realizar as experiências de desempenho usando regras de porta única, e consequentemente apenas uma réplica do BC (que será sempre o responsável por encaminhar os pacotes). Apesar disto, o facto do nosso algoritmo não necessitar de sincronização entre réplicas, faz com que estas experiências sejam uma boa aproximação do desempenho, se o protótipo for implementado usando um *switch* com um melhor suporte às regras multiportas. Para uma comparação mais justa, nas experiências em que outros BCs são avaliados, também são configurados sem replicação.

5.4.2 Comparação das Bibliotecas de Captura de Pacotes

Como foi descrito no capítulo 4, testámos várias bibliotecas para capturar os pacotes enviados pelo cliente e encaminhados pelo BC. As bibliotecas testadas são a *NFQueue* [40] e a *PCAP* [51]. A figura 5.3 mostra que a biblioteca *PCAP* é a que captura mais pacotes por segundo e existe apenas uma diferença máxima de 1% entre bibliotecas. O principal critério de seleção foi o facto da biblioteca *PCAP* fornecer os cabeçalhos *Ethernet* necessários para identificar o componente do sistema que enviou o pacote.

5.4.3 Comparação com Outros Balanceadores de Carga

Esta experiência compara o desempenho do nosso protótipo com outros dois BCs (*httpd-bc* [11] e LVS [52]). Também comparamos com o desempenho dum sistema que não usa

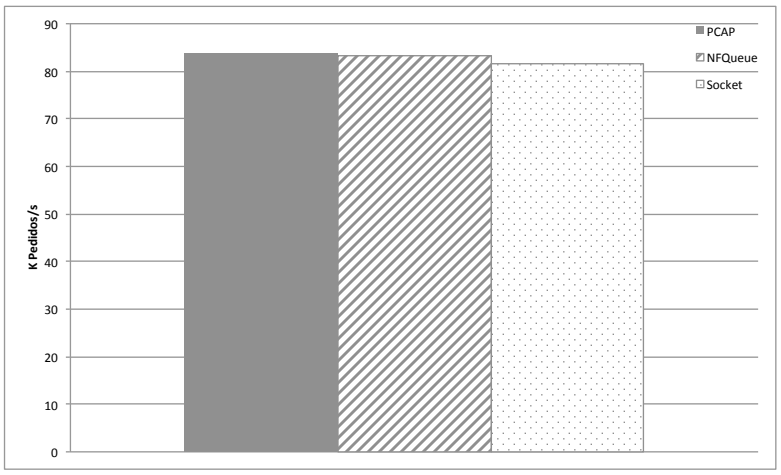

Figura 5.3: Desempenho de várias técncias de captura de pacotes.

BCs para estabelecer uma base de comparação. A figura 5.4 mostra que o desempenho do nosso protótipo é equivalente ao LVS. Isto significa que a nossa arquitetura tem um desempenho similar a um balanceador de carga de nível 4 implementado no *kernel* do sistema operativo. Como esperado, o *httpd-bc* tem um desempenho pior devido ao facto de ser um BC de nível 7 que necessita de estabelecer ligações separadas para os clientes e para os servidores [21, 30].

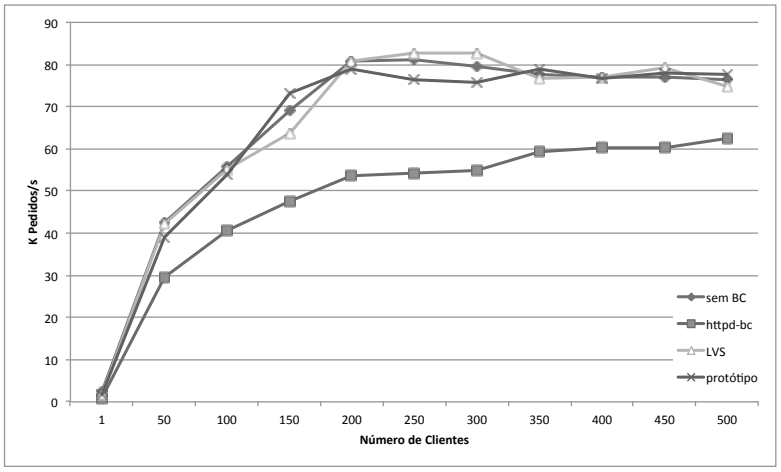

Figura 5.4: Desempenho do protótipo, do LVS e do *httpd-bc* com clientes a enviarem pedidos de 1500 bytes.

Na experiência anterior foram utilizados pedidos de 1500 bytes e respostas que saturavam as ligações da rede (1 Gbit/s). As figuras 5.5 e 5.6 mostram uma experiência similar com um número fixo de clientes (300) e diferentes tamanhos de pedidos e de respostas. Os resultados mostram uma degradação natural no desempenho quando o tamanho dos pedidos e das respostas aumenta, porque este aumento acarreta fragmentação do pedido/resposta em múltiplos pacotes. Os resultados mostram a mesma tendência que

os anteriores: o nosso protótipo e o LVS (BCs de nível 4) apresentam um desempenho similar à base de comparação enquanto que o *httpd-bc* (BC de nível 7) é menos eficiente, especialmente para pedidos e respostas mais pequenos.

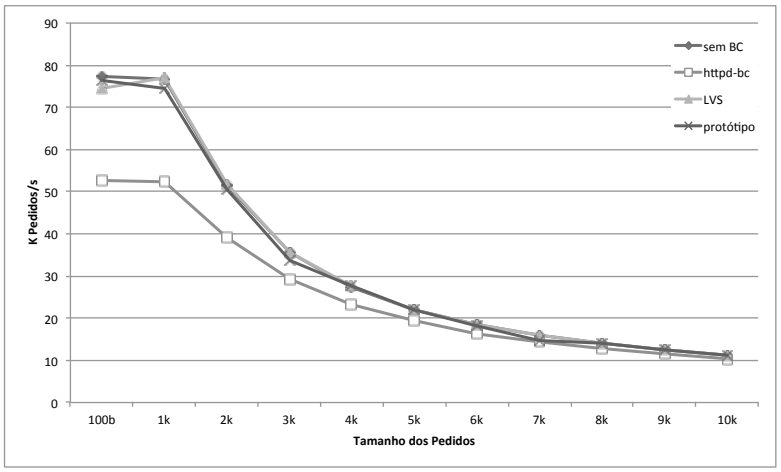

Figura 5.5: Desempenho dos vários BCs para pedidos com diferentes tamanhos.

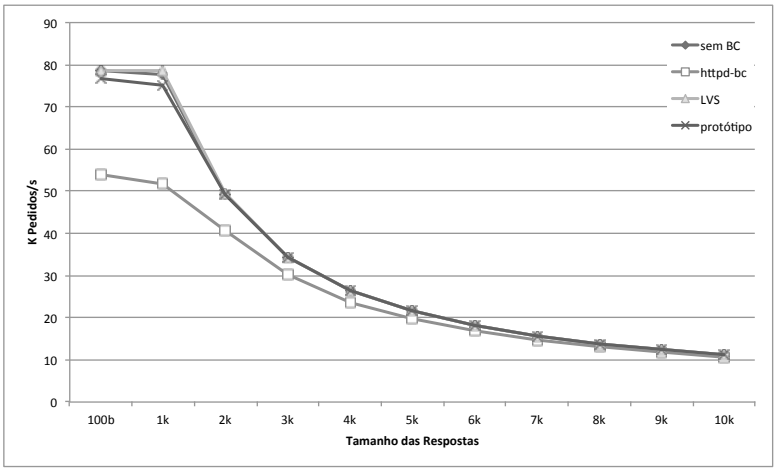

Figura 5.6: Desempenho dos vários BCs para respostas com diferentes tamanhos.

5.5 Escalabilidade

Nesta secção apresentamos resultados que demonstram a escalabilidade do nosso sistema, para isso aumentamos o número de réplicas de servidores e de BCs. Apresentamos também os consumos de CPU dos dois módulos implementados.

5.5.1 Número de Servidores

Nas experiências anteriores considerámos apenas que um servidor respondia aos pedidos. Nestes casos, o nosso protótipo é semelhante às outras soluções populares de BCs. A

figura 5.7 mostra resultados análogos aos resultados anteriores mesmo usando mais servidores (aumentado a capacidade do sistema). O nosso protótipo tem um desempenho semelhante ao LVS e à base de comparação até ao ponto de saturação da rede. Nesta experiência, os testes sem BC foram feitos com 200 clientes a ligarem-se a um servidor diferente.

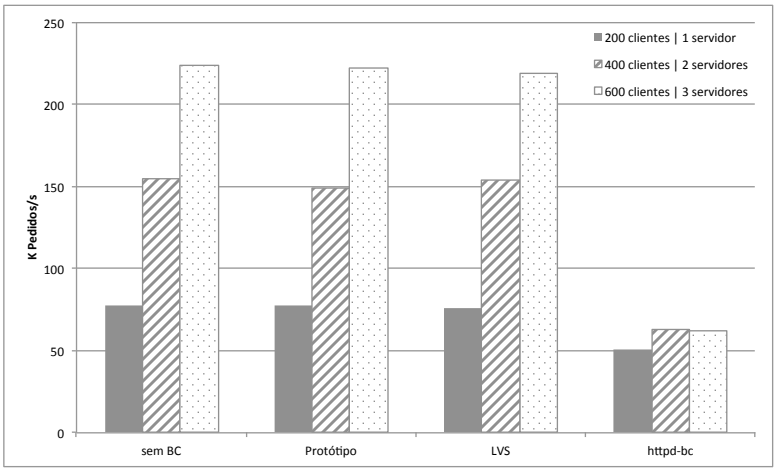

Figura 5.7: Desempenho dos vários BCs quando são adicionados mais clientes e servidores ao sistema (pedidos de 1500 bytes).

A segunda experiência de escalabilidade mostra o custo de executar o módulo do nosso protótipo nos servidores. Para isso, comparamos a utilização de CPU nos servidores em cenários de saturação apenas com o *httpd* e com o *httpd* e o módulo do nosso protótipo. A figura 5.8 mostra os resultados para pedidos pequenos (185 bytes) e pedidos grandes (1500 bytes). Em ambos os casos a figura mostra que o consumo adicional de CPU do módulo é praticamente nulo.

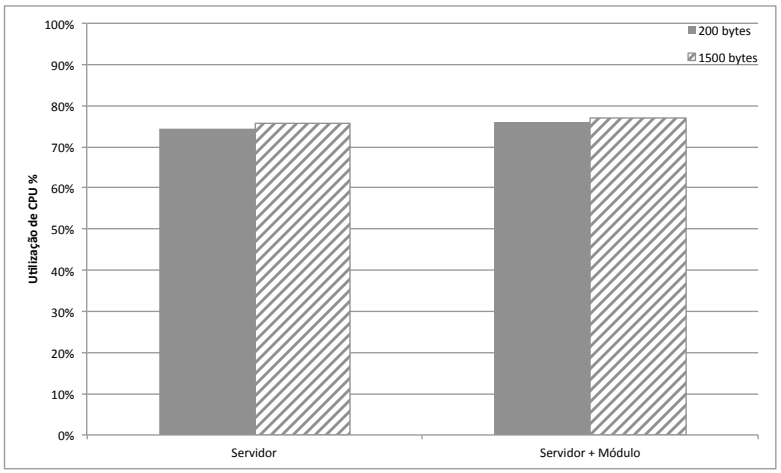

Figura 5.8: Consumo de CPU num servidor *httpd* saturado com e sem o módulo do BC.

5.5.2 Número de Réplicas do Balanceador de Carga

Uma das características interessantes do desenho inexato deste trabalho é a capacidade de escalar o desempenho adicionando mais réplicas ao sistema. Devido às limitações do *switch*, explicadas anteriormente, não é possível escalarmos o desempenho do sistema de forma a mostrar esta característica, por isso realizámos uma experiência com o objetivo de estimar a capacidade máxima de processamento de pacotes por uma réplica do BC nos diversos papéis (responsável, vigia e outro). É de relembrar que "Outro" é o papel em que uma réplica apenas descarta o pacote. Para remover as limitações da rede foi adicionada numa réplica do BC uma aplicação que injeta diretamente pedidos grandes e pequenos de forma a que essa réplica executasse sempre o mesmo papel. A figura 5.9 mostra o desempenho máximo de uma réplica em diferentes papéis.

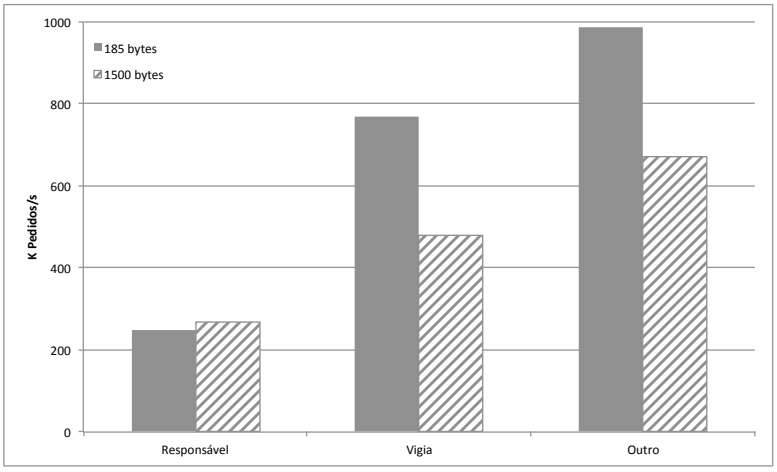

Figura 5.9: Desempenho de uma réplica do BC para os diferentes papéis.

A figura 5.9 mostra que quando consideramos pacotes de 1500 bytes, o responsável encaminha até 270k pedidos por segundo, enquanto que o vigia consegue encaminhar 470k pedidos por segundo (+74% que o responsável), e as outras réplicas, que apenas descartam os pacotes, conseguem processar 650k pedidos por segundo (+141% que o responsável). Esta diferença é similar para pacotes mais pequenos.

Utilizando uma experiência semelhante à experiência em que medimos o consumo de CPU nos servidores, medimos o consumo de CPU de uma réplica BC quando esta assume diferentes papéis. Na figura 5.10 observamos que quando o BC é responsável por encaminhar o pacote, o consumo de CPU é mais elevado, contudo na mesma situação em que os servidores estão a consumir 80% do CPU, a nossa réplica consome apenas perto de 14%.

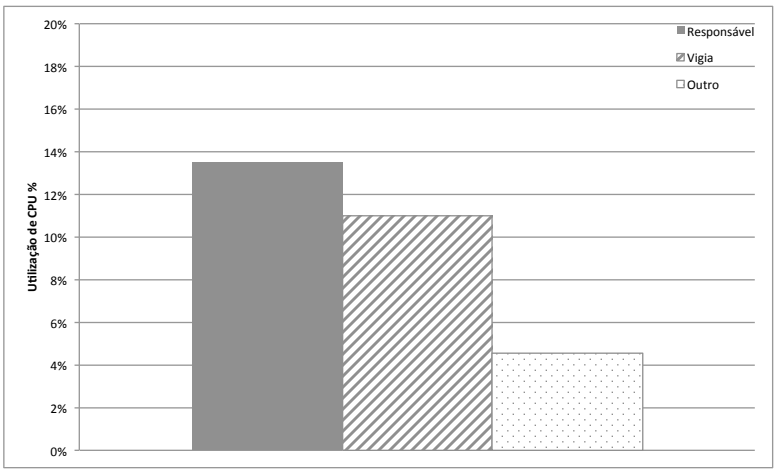

Figura 5.10: Consumo de CPU de uma réplica do BC para os diferentes papéis.

5.6 Funcionamento do Protótipo em Cenários de Faltas

As experiências seguintes avaliam o protótipo em diferentes cenários de faltas. Nestas experiências consideramos $f = 1$, $k = 0$ (3 réplicas de BC), $\texttt{ROUND} = 1$ (cada servidor envia os *bags* para os vigias a cada segundo), $\texttt{TIMEOUT} = 3$ (a retransmissão do pedido ocorre se o pedido não tiver sido encaminhado para o servidor depois de 3 segundos), $\texttt{TH\_ASUSP} = 3$ (um voto de suspeita é enviado para o controlador depois de 3 *bags* sucessivos com erros) e $\texttt{TH\_SUSP} = 100$ (um valor elevado para que não haja impacto nas experiências).

Foram injetados os seguintes comportamentos bizantinos numa réplica do BC:

1. **Paragem:** a réplica incorreta pára;
2. **Rejeição de pacotes:** a réplica incorreta descarta os pacotes;
3. **Corrupção de pacotes:** a réplica incorreta modifica os pacotes antes dos encaminhar para o servidor;
4. **Servidor errado:** a réplica incorreta encaminha os pacotes para um servidor diferente do que é suposto;
5. **Criação de pacotes:** a réplica incorreta cria e encaminha pacotes incorretos para um servidor.

Nestas experiências, sempre que é injetada uma falta numa réplica, ela comporta-se de forma bizantina até ser removida do sistema (falta bizantina sistemática). É de salientar que os mecanismos de confiabilidade usados nos sistemas atuais [3, 7, 52] e estudos recentes [26, 45] apenas recuperam faltas por paragem. Todos os outros comportamentos incorretos necessitariam de uma monitorização avançada que é disponibilizada pelo nosso protótipo.

A figura 5.11 mostra os tempos de detecção e remoção das réplicas sujeitas à injeção dos vários tipos de faltas sistemáticas. O tempo de detecção representa o tempo médio que demora até um vigia suspeitar pela primeira vez que uma réplica está incorreta. O tempo de remoção respeita o intervalo de tempo até $f+1$ vigias informarem o controlador que uma réplica está incorreta.

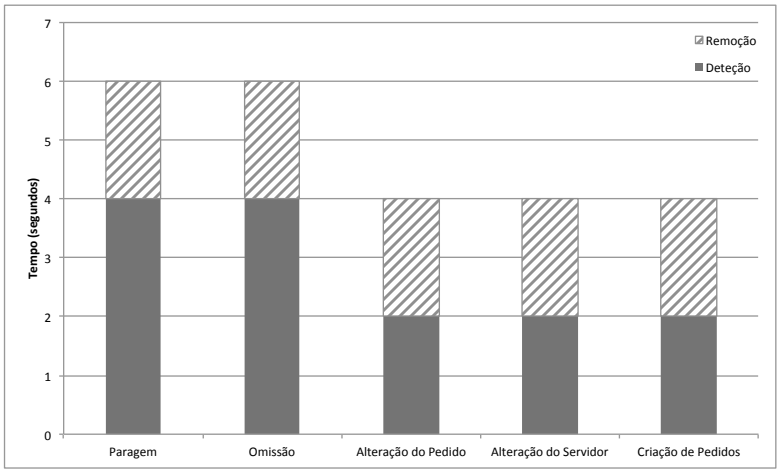

Figura 5.11: Tempos de detecção e remoção de réplicas bizantinas.

Os resultados mostram que as faltas por omissão (paragem e rejeição de pacotes) são as que demoram mais tempo até serem detectadas na nossa configuração: a detecção apenas ocorre depois de `TIMEOUT` $+ 1$ (i.e., 4) segundos (é de relembrar também que um pedido disseminado numa dada ronda apenas será recebido para avaliação na ronda seguinte). Todas as outras faltas (corrupção, servidor errado e criação de pacotes), em que os pacotes são recebidos num *bag* em que não era suposto, são detectadas em dois segundos ($2 \times$ `ROUND`). Para todos os casos são necessárias mais duas rondas (1 segundo cada) para a remoção da réplica incorreta já que o contador de faltas sistemáticas necessita de ser incrementado mais duas vezes para atingir o limite `TH_ASUSP`.

A figura 5.12 mostra a latência média verificada por um cliente quando as diversas faltas são injetadas numa réplica. A figura mostra que as faltas por paragem e a corrupção de pacotes têm um impacto momentâneo na latência verificada pelo cliente, pois os pacotes corretos são recebidos pelos servidores apenas depois dos vigias detectarem a falha (4 e 3 segundos depois, para paragem e corrupção, respectivamente). Como esperado, a criação de pacotes não afeta a latência uma vez que os pacotes do cliente não são afetados por esta falta. Enviando o pacote do cliente para um servidor que não mantém uma ligação com o cliente faz com que o servidor envie um pacote de *reset* (que obriga o cliente a fechar a ligação), fazendo com que não haja nenhum impacto na latência.

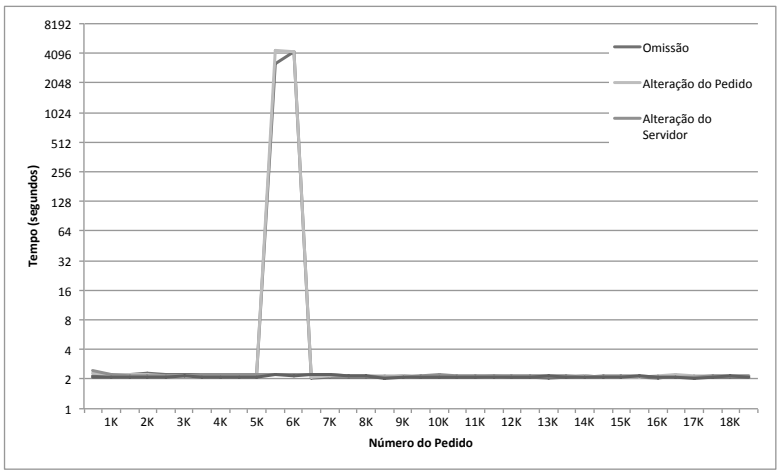

Figura 5.12: Latência verificada pelos clientes em vários cenários de faltas (eixo y em escala logarítmica).

5.6.1 Variação dos Parâmetros de Configuração

Nas experiências anteriores considerámos valores fixos para o `ROUND`, `TIMOUT` e `TH_ASUSP`. A figura 5.13 mostra que ao aumentar o valor do `TIMEOUT`, os tempos de detecção e remoção também aumentam.

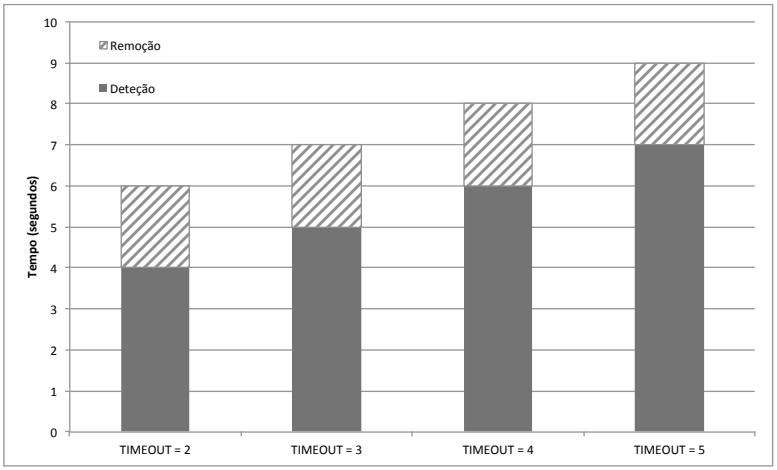

Figura 5.13: Tempos de detecção e remoção de réplicas incorretas para diferentes valores de `TIMEOUT`.

Variar a duração de uma ronda (figura 5.14) também aumenta os tempos de detecção e remoção das réplicas incorretas. O tempo de detecção aumenta porque assumimos $\texttt{TIMEOUT} = 2 \times \texttt{ROUND} + 1$ para evitar que os tempos de espera expirassem antes dos servidores enviarem os *bags* (é de relembrar que o servidor envia o *bag*, com os pedidos processados na ronda i, apenas na ronda $i + 1$). O tempo de remoção aumenta porque são necessárias 3 rondas em que o contador `ASUSP` é incrementado, o que acontece duas rondas após a primeira detecção. Rondas pequenas asseguram que as faltas são detectadas e

removidas rapidamente, mas requerem que os *bags* sejam enviados mais frequentemente, aumentado assim o tráfego transmitido nas ligações entre os servidores e as réplicas do BC. Usando valores grandes ou pequenos para o limite do contador `TH_ASUSP` não faz variar os tempos de detecção (geridos apenas pelo `TIMEOUT`), mas aumenta linearmente os tempos de remoção (figura 5.15).

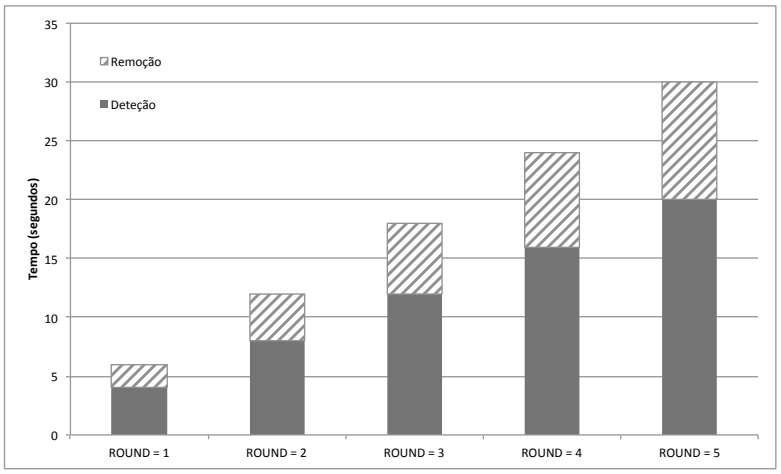

Figura 5.14: Tempos de detecção e remoção de réplicas incorretas para diferentes valores de `ROUND`.

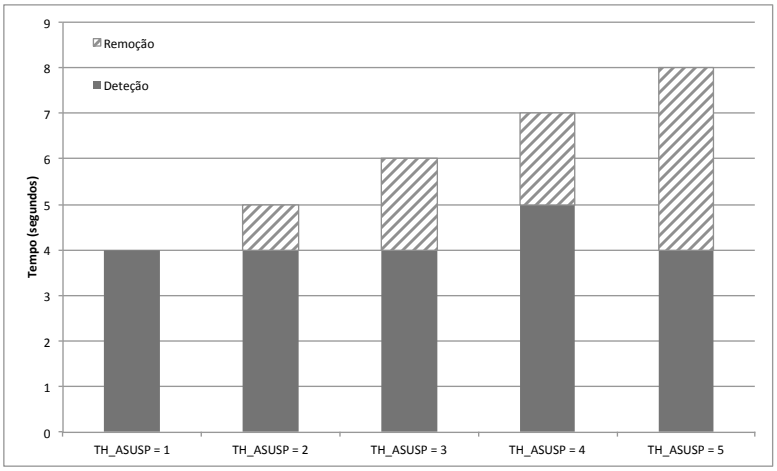

Figura 5.15: Tempos de detecção e remoção de réplicas incorretas para diferentes valores de `TH_ASUSP`.

5.7 Impacto do Desenho Inexato

Para as últimas experiências consideramos o impacto de uma atualização na política de distribuição de carga. Mais precisamente, esta atualização informa as réplicas que foi adicionado mais um servidor e que este deve ser considerado na sua política de distribuição.

Esta atualização, num cenário em que não existe sincronia dos pedidos dos clientes, pode fazer com que as réplicas do BC tomem decisões diferentes, o que pode levar a falsas suspeitas. A figura 5.16 mostra a latência média dos pedidos durante períodos normais e de atualização. Os valores da atualização consistem na latência média durante o segundo de atualização. A atualização é feita sem parar o processamento de pacotes no *switch* (i.e., o passo 1 descrito na secção 3.5.3 não é executado) e sem ignorar rondas (i.e. `IGNORE` $= 0$), ou seja, é o pior caso possível.

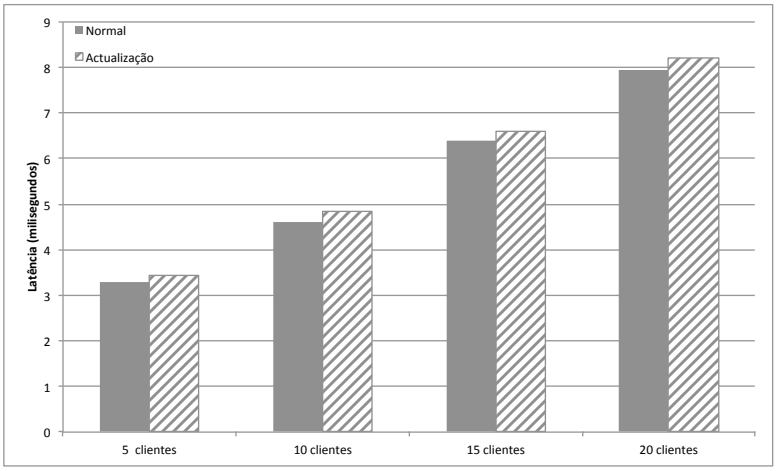

Figura 5.16: Impacto da atualização da política de distribuição na latência do serviço.

Como pode ser observado na figura, a atualização tem um impacto mínimo no sistema. Durante a experiência os valores dos contadores foram incrementados entre 0 e 12 vezes, mas nenhuma remoção foi efetuada (é de relembrar que os contadores também podem ser decrementados quando um BC se comporta corretamente depois de ter sido considerado suspeito).

5.8 Sumário

Nesta secção mostrámos que o nosso protótipo tem um desempenho equivalente a outros BCs usados em situações reais. Mostrámos também a diferença entre as técnicas de disseminação, e que o *switch* com regras multiportas tem um desempenho pior que com regras de porta única.

Mostrámos o comportamento do sistema nos diferentes cenários de faltas bizantinas sistemáticas. Por fim, mostrámos também o algoritmo desenvolvido em funcionamento e como os tempos de detecção e remoção de faltas variam consoante a configuração dos parâmetros. O `TIMEOUT` dos pacotes e o tempo de cada ronda (`ROUND`) alteram o tempo de detecção, e consequentemente, o tempo de remoção. Variar o limite de faltas sistemáticas toleradas (`TH_ASUSP`) altera apenas o tempo de remoção, e o último teste mostra que o impacto do desenho inexato é praticamente nulo.

Capítulo 6

Conclusão

Neste capítulo apresentamos um sumário do trabalho desta dissertação e das suas contribuições. Apresentamos também as limitações do algoritmo e da nossa implementação, e finaliza com algumas ideias para trabalho futuro.

6.1 Sumário dos Resultados

Nesta dissertação apresentámos um BC distribuído e tolerante a faltas bizantinas sistemáticas. Desenvolvemos uma algoritmo de tolerância a faltas bizantinas para ser executado nas réplicas do BC, que detecta e remove os BCs incorretos de uma forma eficiente, sem comprometer o desempenho do sistema. Como consequência, o nosso algoritmo oferece menos garantias de tolerância a faltas, mais precisamente, permite que algumas mensagens incorretas cheguem aos servidores finais (uma característica do paradigma do desenho inexato).

As nossas experiências mostraram que o desempenho do protótipo não replicado equivale ao do LVS, com um total de 80k pedidos de 1500 bytes distribuídos por segundo. Noutras experiências mostrámos que o limite de processamento do nosso protótipo está limitado pela rede de 1 Gbit/s, usada durante as experiências, e que a velocidade de processamento do protótipo era próxima de 8 Gbit/s. Por fim, mostrámos que o algoritmo detecta e remove os BCs incorretos nos diversos cenários de faltas bizantinas sistemáticas, sem grande perturbação na latência.

6.2 Limitações

Dado que não existem trabalhos sobre faltas bizantinas em *middleboxes*, e o trabalho desenvolvido nesta dissertação é um dos primeiros nesta área, existem ainda algumas limitações na nossa implementação:

1. **Faltas por paragem.** Existe um tempo de espera elevado até à detecção das faltas por paragem porque o mecanismo usado para a detecção de faltas é o mesmo usado

para detectar faltas bizantinas sistemáticas, i.e., os vigias esperam um tempo para que os pedidos sejam encaminhados para o servidor, e só após esse tempo é que consideram uma réplica como incorreta.

2. **Desenho Inexato.** Como mencionámos anteriormente, em troca de desempenho os servidores podem receber pedidos corrompidos. Isto deve-se ao mecanismo de detecção das faltas bizantinas necessitar que o servidor Web receba alguns pedidos corrompidos, que são, posteriormente, detectados através dos *bloom filters* enviados pelos servidores para os vigias.

3. **Técnicas de disseminação.** A nossa solução exige que todos os BCs recebam os mesmos pacotes, e para isso é necessário que exista uma técnica que permita disseminar um pacote para todas as réplicas. Explorámos várias técnicas de disseminação de pacotes (*multicast*, *broadcast* e regras *OpenFlow* multiporta), e observámos que as técnicas com melhor desempenho conseguem apenas encaminhar 10% dos pacotes encaminhados sem usar estas técnicas (*unicast*).

6.3 Trabalho Futuro

O trabalho desenvolvido nesta dissertação foi o primeiro passo na introdução de um novo tipo de *middleboxes* que, de forma eficiente, detectam e isolam (através da remoção do componente) faltas por paragem e faltas bizantinas sistemáticas. Mostrámos que é possível implementar um algoritmo de balanceamento de carga eficiente dentro deste paradigma, mas ainda existem muitas áreas para explorar:

- Desenvolver técnicas de disseminação de pedidos mais eficientes para resolver a limitação de desempenho imposta no protótipo.
- Implementar um controlador *OpenFlow* replicado e tolerante a faltas bizantinas.
- Testar o desempenho do protótipo com um *switch* que ofereça uma implementação *OpenFlow* em que as regras usadas pelo protótipo são executadas por *hardware* e não por *software*.
- Comparar o protótipo com os outros BCs em cenários de faltas por paragem.
- Aplicar os conceitos desenvolvidos neste trabalho na concepção de outros tipos de *middleboxes* (por exemplo, *firewalls*).

Capítulo 7

Abreviaturas

DNS	*Domain Name Server*
URL	*Uniform Resource Locator*
MAC	*Media Access Control*
IP	*Internet Protocol*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
HTTP	*Hypertext Transfer Protocol*
ARP	*Address Resolution Protocol*
NAT	*Network Address Translation*
OSI	*Open Systems Interconnection*
SSL	*Secure Sockets Layer*
LVS	*Linux Virtual Server*
DoS	*Denial of Service*
BC	Balanceador de Carga
SITE-E	*Size Interval Task Assignment with Equal Load*
CAP	*Client Aware Policy*
LARD	*Locality-Aware Request Distribution*
WARD	*Workload-Aware Request Distribution*

Bibliografia

[1] S3 data corruption? https://forums.aws.amazon.com/thread.jspa?threadID=22709, 2008.

[2] HP Switch Software - OpenFlow supplement. http://bizsupport2.austin.hp.com/bc/docs/support/SupportManual/c03170243/c03170243.pdf, 2012.

[3] A10 networks ax series. http://www.a10networks.com, 2013.

[4] Apache http server. http://httpd.apache.org, 2013.

[5] Apache' http server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html, 2013.

[6] Cisco load balancing. http://www.cisco.com, 2013.

[7] F5 big-ip. http://www.f5.com, 2013.

[8] Foundry load balancing. http://www.foundrynet.com, 2013.

[9] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Workshop on Hot Topics in Operating Systems*, 2009.

[10] G. Anthes. Inexact design – beyond fault-tolerance. *Communications of the ACM*, 56(4), 2013.

[11] Apache. Apache httpd server load balancer. http://httpd.apache.org/docs/current/mod/mod_proxy_balancer.html, 2013.

[12] Apache. Module connectors for apache httpd. http://tomcat.apache.org/connectors-doc/generic_howto/loadbalancers.html, 2013.

[13] L. Aversa and A. Bestavros. Load balancing a cluster of web servers - using distributed packet rewriting. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference*, pages 24–29, 2000.

[14] L. Bairavasundaram, G. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. In *Proceedings of the USENIX Symposium on File and Storage Technologies*, 2008.

[15] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen. RFC 1945 – Hypertext Transfer Protocol – HTTP/1.0. `http://www.faqs.org/rfcs/rfc1945.html`, 1996.

[16] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the ACM/EuroSys Conference on Computer Systems*, 2008.

[17] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[18] L. Borzemski and K. Zatwarnicki. A fuzzy adaptive request distribution algorithm for cluster-based web systems. In *Parallel, Distributed and Network-Based Processing*, pages 119–126, 2003.

[19] T. Bourke. *Server load balancing*. O'Reilly & Associates, Inc., 2001.

[20] T. Brisco. DNS Support for Load Balancing. RFC 1794, 1995.

[21] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Survey*, 34(2):263–311, 2002.

[22] E. Casalicchio and M. Colajanni. A client-aware dispatching algorithm for web clusters providing multiple services. In *Proceedings of the International Conference on World Wide Web*, pages 535–544. ACM, 2001.

[23] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 2002.

[24] M. Correia, D. Ferro, F. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[25] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available web server. In *Proceedings of the IEEE Computer Conference*, pages 85–92, 1996.

[26] P. Patel et al. Ananta: Cloud scale load balancing. In *ACM Special Interest Group on Data Communication*, 2013.

[27] International Organization for Standardization ISO. Information technology - open systems interconnection - basic reference model: The basic model. Technical report, 1994.

[28] J. George, B. Marr, B. Akgul, and K. Palem. Probabilistic arithmetic and energy efficient embedded signal processing. In *IEEE/ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2006.

[29] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM Special Interest Group on Data Communication*, 2011.

[30] K Gilly, C. Juiz, and R. Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, 14(2):105–131, 2011.

[31] M. Harchol-Balter, M. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. In *Proceedings of the International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, pages 231–242, 1998.

[32] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network dispatcher: A connection router for scalable Internet services. In *Proceedings of the International Conference on World Wide Web*, 1998.

[33] A. A. Hwang, I. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[34] IBM. Websphere application server, load balancer administration guide. Technical report, IBM, 2006.

[35] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, 1982.

[36] Chiang M.L. Liu, H.H. Tcp rebuilding for content-aware request dispatching in web clusters, 2005.

[37] M. Luo, C. Yang, and C. Tseng. Analysis and improvement of content-aware routing mechanisms. *Institute of Electronics, Information and Communication Enginners Transactions on Computer Systems*, (1):227–238, 2005.

[38] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks.

Special Interest Group on Data Communication: Computer Communication Review, 38(2):69–74, 2008.

[39] Netfilter. Iptables. `http://www.netfilter.org/`, 2013.

[40] Netfilter. Nfqueue. `http://www.netfilter.org/projects/libnetfilter_queue/`, 2013.

[41] E. Nightingale, J. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer pcs. In *EuroSys*, 2011.

[42] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the International Conference on Architectural support for programming languages and operating systems*, pages 205–216. ACM, 1998.

[43] K. Palem, L. Chakrapani, Z. Kedem, A. Lingamneni, and K. Muntimadugu. Sustaining moore's law in embedded computing through probabilistic and approximate design: Retrospects and prospects. In *IEEE/ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2009.

[44] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *ACM Internet Measurement Conference*, 2013.

[45] S. Rajagopalan, D. Willians, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *ACM Symposium on Cloud Computing*, 2013.

[46] S. Rajagopalan, D. Willians, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[47] T. Roeder and F. B. Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems*, 2010.

[48] S. Jain et al. B4: Experience with a globally-deployed software defined WAN. In *ACM Special Interest Group on Data Communication*, 2013.

[49] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *ACM Special Interest Group on Data Communication*, 2012.

[50] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4), 2010.

[51] Tcpdump/Libpcap. Pcap library. `http://www.tcpdump.org`, 2013.

[52] W. Zhang. Linux virtual server for scalable network services. In *Prooceedings of the Ottawa Linux Symposium*, 2000.